Test Case Generation for Flexible Real-Time Control Systems

Nilsson, Robert; Henriksson, Dan

Published in:
Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on

2005

Test Case Generation for Flexible Real-Time Control Systems

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

Dan Henriksson
Department of Automatic Control
Lund University
Box 118, SE-22100 Lund, Sweden
dan.henriksson@control.lth.se

Abstract

Temporal correctness is crucial for the dependability of real-time control systems. A problem with testing such systems is the dependency on the execution orders of tasks. Mutation-based testing criteria have been proposed to determine which execution orders need to be exercised to verify that real-time systems are timely. For flexible control systems, timeliness in itself may only be relevant for a sub-set of tasks, whereas maintained control performance in the presence of worst-case jitter and disturbances is essential. This paper presents an extension to the co-simulator tool TrueTime, to support mutation-based testing of control performance and timeliness. Further, an approach for automatic generation of test cases using genetic algorithms is presented. A conclusion is that testing criteria for timeliness can be used to increase confidence in the dependability of flexible control systems.

1. Introduction

Current real-time control systems must be both flexible and dependable. On the other hand, there is a desire to increase the number of services that real-time systems offer while using few, off-the-shelf hardware components. This increases system complexity and introduces sources of temporal non-determinism. Thus we need methods to detect violation of timing constraints and poor control performance using computer architectures where we cannot to rely on accurate off-line assumptions.

Timeliness is the ability for software to meet timing constraints. For example, a timing constraint can be that it should never take more than 100 ms between an alarm is activated until a robot arm enters a safe state. If system timeliness is violated, a *timeliness failure* has occurred.

Response times of concurrent tasks depend on the order in which the tasks execute. This is particularly evident in event-triggered and dynamically scheduled systems because sporadic interrupts can continuously influence the execution order and schedule. Also, hardware caches affect the execution times of tasks, causing response times to become non-deterministic with respect to the inputs, and thus, complicate verification and accurate estimations.

Timeliness of embedded real-time systems is traditionally analyzed and maintained using scheduling analysis or regulated online through admission control and contingency schemes [19]. However, such techniques make assumptions about the tasks execution behavior and request patterns. Further, doing full scheduling analysis with non-trivial system models is complicated. Thus, analysis must be complemented with timeliness testing.

Many real-time systems have tasks that implement control applications. Such applications interact with physical processes through sensors and actuators to achieve a control goal. For example, a painting robot may have a control application that periodically samples joint angles and sets different motor torques so that the robot movement becomes smooth and aligned with the painted object.

We use the concept *flexible control system* to denote a real-time system that is event-triggered and dynamically scheduled and has a mix of reactive *hard* and *soft* tasks. The *hard* tasks must always meet their timing constraints, whereas *soft* tasks are more tolerant to delay and irregularities, and typically some deadlines of soft tasks can be missed before the system fails.

Most control applications are based on feedback principles, which means that they are inherently robust against occasional timeliness failures. An occasional deadline miss for a periodic controller task is not fatal for system stability, but can rather be seen as a disturbance acting on the control system. Hence, controllers can be implemented with soft tasks. Soft tasks are also referred to as *adaptive* tasks [5], in that missing single deadlines does not jeopardize correct system behavior, but only leads to a performance degradation. One example is EDF scheduling during overload, which will effectively lead to rescaling of the sampling periods of all tasks. In this case actually all tasks will miss all their deadlines, however, the performance of the control loops may still be acceptable with the slightly longer sampling intervals.

Instead, control algorithms contain built-in timing constraints that are more subtle than response time deadlines.

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

The delay in each sample between the readings of the inputs and the generation of the outputs is known as *input-output latency*. Excessive input-output latency will compromise the performance of the control system, and may even cause instability. Further, depending on the process under control and the controller design, there will be a maximum variation (*jitter*) in the sampling instants and the input-output latency that can be tolerated to guarantee system stability [6].

If these constraints are violated, the control application may fail its control goal or even become unstable. We call this a *control failure*. Moreover, since faults in the estimation of temporal properties may result in unanticipated behavior, it is relevant to test control performance of the soft controller tasks.

In this paper we present an extension to the real-time co-simulation tool TrueTime to support test case generation. We also evaluate the capability of revealing failures in flexible control systems in a proof-of-concept test case generation experiment. Our results indicate that a test case generation method for testing of timeliness can also be used for generating test cases for revealing control failures.

2. Automated test case generation

When testing software, a *test criterion* is typically set up to define the test requirements that must be satisfied. Examples of test criteria include 'execute all statements' and 'cover all transitions' in a state machine.

The mutation-based testing technique presented in this paper is mainly inspired by a specification-based method for automatic test case generation presented by Ammann Black and Majurski [2]. The main idea behind the technique is to systematically "guess" what faults a real-time design contains and then evaluate what the worst effect of such faults could be.

Each hypothesized fault is represented as a copy of the system specification containing that fault; such a specification is called a *mutant*[*]. As a part of test case generation, the mutant models are analyzed and classified as benign or malignant. Mutants containing faults with bad consequences are classified as *malignant* and specialized test cases are constructed that aim to reveal those faults if they exist in the final implementation.

For a more detailed overview of mutation-based test case generation, consider Figure 1. The inputs to mutation-based testing are a specification of a real-time system and a test criterion. The test criterion specifies the mutation operators to use when creating mutants, and thus, determines the kind of test cases that are produced. An advantage of using mutation-based testing criteria is that the testing effort can be estimated and quantified by the number of malignant mutants.

[*]Note : These mutants are not related to the mutations and cross-overs performed when using genetic algorithms

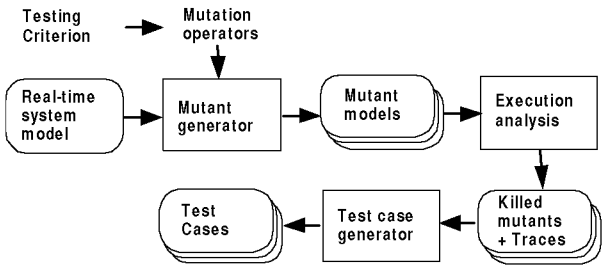

Figure 1. Mutation-based test generation

A mutant generator applies the mutation operators and sends the mutated specifications to an execution order analyzer that determines if and how a mutation can lead to a timeliness or control failure (see Figure 1). If the analysis reveals a missed hard deadline or an unstable controller, it is marked as *killed*, otherwise the mutation is considered to be benign and discarded. For pure timeliness testing, this execution order analysis can be done using model checking [14].

Traces from the killed mutants are sent to a test case generation filter that converts the traces to input sequences and their corresponding expected and critical execution orders. These are used as test cases for the target system.

Test execution is focused on running the generated input sequences and trying to detect the derived critical execution orders using, for example, prefix-based or non-deterministic testing techniques [10, 13, 17].

2.1. System model

Real-time application behavior is typically modelled by a set of periodic and sporadic tasks that compete for system resources. *Periodic* tasks are requested with fixed inter-arrival times, thus the times when the task will be requested are known. *Sporadic* tasks can be requested as a response to some event at any time. However, to simplify analysis, sporadic tasks are specified with a *minimum inter-arrival time*. If no minimum inter-arrival time can be determined, the task is *aperiodic*. Each real-time task has a specified *deadline* and sometimes an *offset*, which denotes the time before any task of that type is requested.

In this paper we use a subset of Timed Automata with Tasks (*TAT*) [15, 7] to specify the assumptions about the system under test.

Timed Automata (*TA*) [1] have been used to model different aspects of real-time systems. A timed automaton 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, e.g., a time constraint. An *action* can perform operations such as assigning 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. Within *TAT*, *TA* is used for specifying *activation pattern* of tasks, i.e., the points

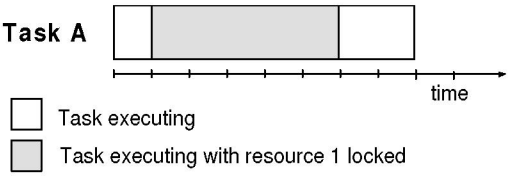

Figure 2. Visualization of execution pattern

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

Table 1. Example TAT task set description

in time when a task is requested for execution. In this paper we focus on sporadic and periodic tasks with generic automata templates.

TAT extends the *TA* notation with a set of real-time tasks P. The set P represents tasks that perform computations in response to requests. Elements in P express information about tasks as quadruples $(c,\ d,\ SEM,\ PREC)$. c is the required execution time and d is the relative deadline. These values are used to create a new task instance as it is released by a *TAT* automaton action. Shared resources are modelled by a set of system-wide semaphores R. SEM is a set of tuples of the form $(s,\ t_1,\ t_2)$ where t_1 and t_2 are the relative lock and unlock times of semaphore $s \in R$ when an instance of the task is executed. *Precedence constraints* are relations between pairs of tasks A and B stating that an instance of task A must have completed between the execution of two consecutive instances of task B. For example, such constraints can model a blocking producer-consumer relation between task A and B. Hence, $PREC$ is a subset of P that specifies which tasks must precede a task of this type.

A specification of the execution of a task, including the points in time when resources are locked and unlocked, is called an *execution pattern* in this paper. Figure 2 shows the execution pattern of task A in Table 1.

In *TAT*, task execution times are fixed. This may appear unrealistic if the input data to a task is allowed to vary. However, to divide the testing problem, this test case generation step assumes that each task is associated with a particular (typical or worst-case) equivalence class of input data so that the only variance in execution times comes from non-deterministic components and the target platform. Several complementary methods exist for deriving such classes of input data for real-time tasks [16, 12]. Further, when a malignant mutant is found, tasks can be run in a critical execution order to see if a failure can be reproduced in the real system using other input data.

2.2. Mutation operators

A mutation-based test criterion is defined by a set of mutation operators. Mutation operators have previously been presented for testing of timeliness and formally defined for *TAT*-specification models [14]. In this paper, we summarize the relevant operators informally and discuss the faults generated by the operators from a flexible control system perspective. 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 operators: Execution time mutation operators increase or decrease the assumed execution time of a task by a constant Δ. These mutants represent an overly optimistic estimation of the worst-case (longest) execution time of a task or a overly pessimistic estimation of the best-case (shortest) execution time. Estimating execution times is generally very hard [16]. 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.

Lock time operators: Lock time mutation operators increase or decrease the time when a particular resource is locked relative to the start of that task. In one mutant the lock time is increased with Δ and in the other mutant the lock time is decreased by Δ. 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. This mutation operator requires test cases that can distinguish an implementation where a resource is locked too early from one where it is not.

Unlock time operators: Unlock time mutation operators change the 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. This mutation operator requires test cases that can distinguish an implementation where a resource is held too long from one where it is not.

Inter-arrival time operators: This operator decreases or increases the inter-arrival time between requests for a task execution by a constant time Δ. This reflects a change in the system environment that causes requests to be more frequent than expected. The resulting test cases will stress the system to reveal its sensitivity to higher frequencies of requests. For periodic tasks (E.g., controllers) a decrease in invocation frequency may also result in failures.

Pattern offset operators: Recurring requests can have patterns that are assumed to have fixed offsets relative to each other; for example, periodic tasks with harmonic activation patterns. This operator changes the offset between such patterns by increasing or decreasing the offset with Δ time units.

3. Flextime : A test generation extension

Flextime is an add-on tool for the real-time control systems simulator TrueTime [8]. The purpose of the Flextime add-on is primarily to support automated analysis and mutation-based test case generation. For this purpose TrueTime must be adapted to *(i)* do efficient simulation of *TAT* system models, *(ii)* support structured parametrization of simulations, and *(iii)* simplify extensions that are consistent with *TAT* specifications.

When Flextime is used for mutation-based test case generation, *TAT* models should be mapped to simulation entities. The following subsections describe the TrueTime tool and how *TAT* task sets and activation patterns are mapped to simulations by the Flextime extension.

3.1. TrueTime

TrueTime is a real-time kernel simulator based on MATLAB/Simulink. The main feature of the simulator is that it offers the possibility of co-simulation of task execution in a real-time kernel and continuous-time dynamics modeling controlled plants. The simulator is mainly used for integrated design of controllers and schedulers, and can be used to analyze the effects of timing non-determinism on the performance of the control systems.

The TrueTime kernel is flexible and highly configurable. Both periodic and aperiodic tasks are supported, and the attributes of the tasks may be changed dynamically during simulation. The scheduling algorithm used by the kernel is configurable by the user. Synchronization between tasks is supported by events and shared resources can be protected with mutual exclusion monitors.

Each task in TrueTime is implemented in a separate code function that defines the execution behavior. The code function includes everything from interaction with resources and I/O ports and networks to specification of execution time of different segments. The TrueTime code functions may be written either as C++ functions or as MATLAB m-files.

3.2. Task sets and execution patterns

For the purpose of automated analysis and mutation-based test case generation, we find it useful to separate between application functionality and execution behavior. Therefore, in the Flextime extension, execution times, resource requirements, and precedence constraints are specified separately from code functions. This specification style makes it possible to specify execution patterns of large task sets without having to generate a specific code function for each type. The role of code functions in Flextime is specialized to perform control-related calculations and to interact with external Simulink blocks.

Figure 3 shows a subset of the class diagram of Flextime. The class *ftTask* is an abstract class that maps down to the TrueTime tasks. This means that when objects of any of the sub-classes to this class are created, a TrueTime task is also created and initialized. The abstract *ftTask* class contains basic information about tasks, such as periods, deadlines and offsets. Moreover, the *ftTask* class extends TrueTime tasks with a list of execution items that defines the execution pattern for each instance of this task.

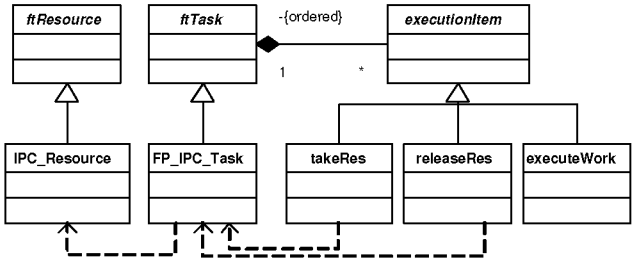

Figure 3. Flextime classes

The sub-classes of *ftTask* and *ftResource* are primarily used for supporting different concurrency control protocols, but other types of execution environment extensions are also supported. For example, one pair of sub-classes can be used to simulate tasks and resources under the immediate priority ceiling protocol [18], whereas another pair may be used for simulation of tasks under EDF scheduling and the stack resource protocol [4]. The reason why sub-classes are needed for both types of entities is that such protocols often require specific data to be kept with task and resource representatives.

When an *ftTask* begins its execution, a virtual *do_seg* method is called sequentially on each item in the execution item list. Execution items of type *takeRes* and *releaseRes* specify that a particular resource is to be locked or unlocked. The *do_seg* function in these execution items simply invokes a corresponding virtual *take* and *release* function in the *ftTask* class with the resource identifier as a parameter. In this way, the logic associated with acquiring and releasing resources can be implemented in the protocol-specific sub-classes of *ftTask*, and execution item classes remain protocol independent. Execution items of type *executeWork* are generic and specify that execution of code should be simulated for some duration, and optionally, that a segment of a Flextime code function should be executed.

3.3. Activation patterns

The activation patterns from environment automata triggering periodic tasks are deterministic and can simply be included in the static configuration of the simulator. The activation pattern for sporadic tasks should be varied for each iteration of the simulation to find execution orders that can lead to timeliness or control failures.

Consequently, an input to the simulation of a particular system (corresponding to a *TAT* model) is the activation patterns for the sporadic tasks. The relevant output from the simulation is an execution order trace where the sporadic requests has been injected according to the activation pattern. A "positive" output from a mutation testing perspective is an execution order trace that contains violated time constraints or simulated control failure.

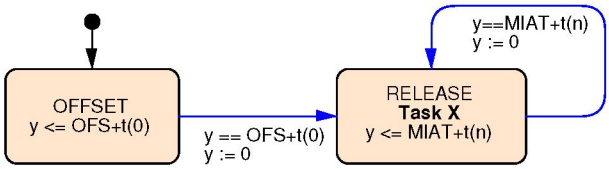

Figure 4. Annotated TAT template

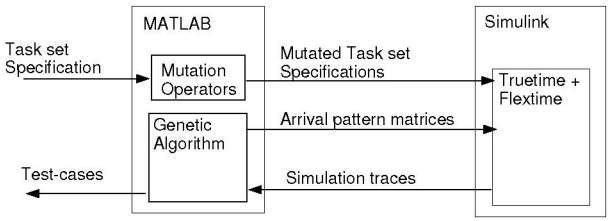

Figure 5. Flextime tool

By treating test case generation like an optimization-based search problem, different heuristic methods can be applied in order to find a feasible activation pattern for revealing failures. In Section 5 we present an experiment where genetic algorithms are used for this purpose. Genetic algorithms have previously been used on various noncontinuous search problems [11], and for testing other aspects of control systems [20].

Figure 4 contains an annotated *TAT*-automaton for describing activation patterns of sporadic tasks. The template has two parameters that are constant for a particular mutant. The constant OFS denotes the assumed minimum offset, i.e., the minimum delay before any instance of this task can be requested. The constant MIAT denotes the assumed minimum inter-arrival time between instances of this task. An array of delay values $t(0..m)$ defines the variable part of the intervals between requests of this task. The constant m is the maximum number of arrivals that can occur in the simulated interval. By combining the arrays for all sporadic tasks we get a matrix $t(1..s, 0..m)$ of real values, where each row corresponds to an activation pattern of a sporadic task. Flextime supports importing activation pattern matrices of this type from the global MATLAB workspace. Moreover, relevant information from the simulation run is logged and exported to MATLAB where it can be analyzed, filtered and converted to test cases.

4. Using Flextime for test generation

Figure 5 gives an overview of how Flextime is used together with other tools to perform automated test case generation. As seen in the figure, a task-set specification must be supplied as input to Flextime simulations and mutation operators. Task-set parameters and execution item lists can be initialized in two different ways in the Flextime tool. One way is to define the execution item lists and task set parameters statically in the TrueTime initialization code. Figure 6 shows the C++ syntax required for initializing the task-set of Table 1. If no specific C++ initialization file is given, Flextime assumes that the system

ID	TS matrix				
	Type	Priority	Period /MIAT	Offset /OFS	Deadline
A	SPOR	1	0.040	0.0	0.020
B	PER	2	0.040	0.020	0.020

Table 2. Initialization matrix TS

ID	XP matrix					
A	0.001	-1	0.005	-1	0.002	NaN
B	0.002	-1	0.006	-1	0.001	NaN

Table 3. Initialization matrix XP

characteristics for a simulation is given through two matrices, TS and XP, in the global MATLAB workspace.

As seen in Table 2, the TS matrix contains one row for each task, specifying its type, priority, period, offset, and deadline. Depending on the type of the task, some fields are interpreted differently. For example, if the type field specifies a hard sporadic task, then the period and offset fields are interpreted as the values for the MIAT and OFS parameters for the template in Figure 4.

The rows in an XP matrix contain the execution patterns for the tasks with the same row number in the TS matrix. All positive values are translated to simulated execution time. All negative values are assumed to be integers and are used for locking and unlocking the shared resources with the specified index. The XP matrix for the task set in Table 1 is given in Table 3. The first occurrence of '-1' in Table 3 means that the resource with index '1' should be locked, whereas the second occurrence means that it should be unlocked.

The matrix representation of task sets has the advantage that different mutation operators easily can be applied to create new mutants. If new task types, concurrency control mechanisms, or scheduling protocols are used, the C++ initialization file must be customized accordingly.

Returning to Figure 5, the task set specification is used as input to mutation operators, which automatically create mutants containing hypothesized faults. For the purpose of the following experiments, these operators have been implemented for the matrix representation.

4.1. Applying genetic algorithms

For each created mutant, a search is performed to find an activation pattern that forces the mutant to miss a hard deadline or causes a control failure. A genetic algorithm drives the simulation of a particular mutant by providing a *population* of initially random activation pattern matrices as input. For each activation pattern matrix, the execution order and control performance traces from the simulation are used to calculate a fitness value. The fitness value reflects the ability of the activation pattern to show a bad behavior of the mutant.

Based on their fitness value, the best activation patterns are copied and changed according to stochastic heuristics. The newly created activation patterns replace some of the

least optimal activation patterns in the set and the evaluation is iterated in a new *generation* . This way, the different execution orders of a mutant are searched for missed deadlines and bad control performance. Consequently, an application-specific fitness function must be provided to use genetic algorithms.

For testing of flexible controllers, both potential timeliness violations and poor control quality must be used to calculate a good fitness value to drive the heuristic search. The *slack* of a real-time task is the time between the actual response time of a task instance and its absolute deadline. For the timeliness factor, the minimum slack among hard tasks provides an intuitive fitness value. All slack times can be recorded during simulation.

For evaluation of control performance it is common to use weighted quadratic cost functions. For a scalar system, with one output y and one input u, the cost can be written

$$J = \int_0^{T_{sim}} \left(y^2(t) + \rho u^2(t)\right)\, dt \qquad (1)$$

where the weight factor, ρ, expresses the relation between the two counteracting objectives of control design, i.e., to keep the regulated output close to zero and to keep the control effort small. The controllers designed for the inverted pendulum control described in the next section are explicitly designed to minimize a cost function of the type given by Equation (1). This is called LQ control [3].

The higher the cost during a simulation run, the worse the control performance. Therefore, in this context, we assume that $1/J$ is proportional to the control performance.

The fitness of a simulation trace is defined as

$$F = \sum_k \frac{1}{J_k} - S_{min} * w, \qquad (2)$$

where S_{min} denotes the least slack observed for any hard real-time task. The variable J_k denotes the value of J for flexible controller k at the end of the simulation. A weight variable w is used for adjusting the minimum slack so that the timeliness factor is of the same magnitude as the control quality factor.

Apart from calculating the general fitness that drives the genetic algorithm heuristics towards evaluating more optimal solutions, the fitness function can also be used to detect failures and halt the search. Relevant failure conditions are that *(i)* a hard critical deadline is missed, *(ii)* the control system becomes unstable (the cost, J, exceeds some threshold value), or *(iii)* a control constraint is violated, for example, the motion of a robot arm becomes too irregular. Failure condition *(i)* and *(ii)* can easily be detected by checking the minimum slack of hard tasks and the value of the cost function for the controller tasks. Failure condition *(iii)* is application-specific and might require specific values to be traced during simulation.

```
// String ID
IPC_Resource R1("State_Sem");

// TYPE, StringID, Priority , MIAT, OFS, Deadline
FP_IPC_Task A(SPOR,"Safety_check", 1 , 0.040,0.0,0.020);
A << 0.001 <<+R1<< 0.005 << -R1 << 0.002 <<  FINISHED;

FP_IPC_Task B(PER,"Pendulum", 2 , 0.040, 0.020, 0.040);
B << 0.002 << +R1 << 0.006 << -R1 << 0.001 << FINISHED;
```

Figure 6. C++ Syntax for initializing task-set

5. Proof-of-concept experiment

The purpose of this experiment is to investigate if a mutation-based testing technique can generate test cases for revealing timeliness and control failures in flexible control systems. Hence, the experiment should evaluate whether the mutation operators can create malignant mutants and how effective our genetic algorithm based tool set is in finding such malignant mutants.

For this experiment we simulate a real-time system with fixed priorities and shared resources under the immediate priority ceiling protocol [18]. The task set consists of three soft periodic tasks that implement flexible controllers for balancing three inverted pendulums. The linearized equations of the pendulums are given as

$$\ddot{\theta} = \omega_0^2\theta + \omega_0^2 u \qquad (3)$$

where θ denotes the pendulum angle and u is the control signal. ω_0 is the natural frequency of the pendulum. The controllers were designed using LQ-theory with the objective of minimizing the cost function

$$J = \int \left(\theta^2(t) + 0.002u^2(t)\right)\, dt \qquad (4)$$

Further, the system has four sporadic real-time tasks with hard deadlines, assumed to implement logic for responding to frequent but irregular events, for example, external interrupts or network messages. The system also has two resources that must be shared with mutual exclusion between tasks. Examples of resources are data structures containing shared state variables and non-reentrant library functions.

Table 4 lists the exact properties of the simulated task set. The first column ('ID') contains task identifiers. Columns two to five contain the *TAT* task set description tuple as described in Section 2.1. The column 'IAT' contain the assumed inter-arrival times of tasks; periodic tasks are released with fixed inter-arrival times and the minimum inter-arrival times of the sporadic tasks are defined using the 'MIAT' parameter in Figure 4. Column 'OFS' contain the corresponding parameter value for the sporadic task template; for periodic tasks, this column contains the offset.

ID	c	d	SEM	PREC	IAT	OFS
A	3	7	{(R2,0,2) }	{}	≥ 30	10
B	5	15	{(R1,0,3)(R2,2,5)}	{}	≥ 40	20
C	4	20	{(R1,0,3)}	{}	≥ 40	0
D	5	26	{(R2,2,5)}	{}	≥ 50	28
E	5	20	{(R1,4,5)}	{}	20	1
F	5	29	{ (R2,0,4) }	{}	29	1
G	5	35	{ (R1,0,3)}	{}	35	1

Table 4. Case study task set

Failure type ⇒		Timeliness			Control		
Mutation operator	**T**	**K**	**A**	**G**	**K**	**A**	**G**
Execution time	14	1	1.0	2	6	5.2	29
Lock time	13	2	2.0	3	0	0	-
Unlock time	15	1	1.0	2	0	0	-
Inter-arrival time	14	0	0	-	5	2.1	16
Pattern offset	13	0	0	-	0	0	-
Total	69	4	4.0	-	11	7.3	-

Table 5. Mutants killed in case study

Three continuous-time blocks modeling the inverted pendulums were included in the simulation and connected in a feedback loop to the TrueTime block with the flexible control system. Each pendulum has slightly different natural frequency, ω_0, and the goal of the control application is to balance the pendulums to an upright position. The pendulums have an initial angle of 0.1 radians from the upright position when the simulation starts. An application-specific control failure is assumed to occur when the angle of a pendulum becomes greater than or equal to $\pi/8$ (~ 0.39) radians.

A set of mutants was generated by applying the mutation operators described in Section 2.2 on the extended task set in Table 4 using a Δ of two time units for the first three mutation operator types and four time units for the last two. The total number of mutants generated for each operator type is listed in column 'T' of Table 5. The genetic algorithm toolbox [9], developed at North Carolina University was used to construct a genetic algorithm that could interact with the Flextime tool.

For the genetic algorithm setup, we used a population size of 25 activation pattern matrices. A mix of generic cross-over functions supplied with the genetic algorithm toolbox and heuristic cross-over functions customized for revealing timeliness faults was used for stochastically changing activation patterns. The fitness function defined in Section 4.1 was used when analyzing the simulation traces.

First, the unmodified system was simulated for 200 generations to gain confidence in the assumed correct specification. This was repeated five times with different random seeds to protect against stochastic variance. No failures were detected in the original model.

Second, each mutant was simulated for 100 generations or until a timeliness or control failure was detected. When a mutant was killed, the same activation pattern was applied on the assumed correct model. The motivation for this extra step is to further increase the confidence in the correctness of the specification model.

The experiment was repeated five times to assess the reliability of the approach. Table 5 summarizes the results for each mutation operator and failure type. The number of mutants that was classified as malignant in any of the experiments is listed in columns marked "K". Columns marked "A" lists the average number of malignant mutants that were killed per experiment. The average number of generations needed to kill malignant mutants is listed in column "G".

As seen in Table 5, our mutation-based approach that uses the Flextime tool automatically generate test cases for revealing both timeliness and control failures.

Further, the malignant mutants that cause timeliness failures were killed in all of the experiments. This result indicates that the genetic algorithm is effective in revealing critical execution orders in flexible control systems of this size. The low average of generations needed to reveal these failures suggests that many execution orders lead to failures in the malignant mutant specifications.

The relatively low average of killed mutants causing control failures indicates that finding a critical scenario with respect to control is more difficult. A possible explanation is that the optimization problem contains local optima with respect to control performance fitness.

A possible way to increase the reliability is to redo the search multiple times using a fresh initial population. Since the approach for searching the mutant specifications is fully automated, the additional cost of searching multiple times may be acceptable.

Lastly, for this system we actually observe a relatively large number of malignant mutants that lead to control failures. This result suggests that mutation operators for testing of timeliness indeed is useful for testing control performance.

6. Conclusions

This paper has presented an extension to the real-time co-simulator TrueTime that prepares it for interacting with heuristic search algorithms for generation of test cases. The extension tool maps configurable *TAT* task set specifications to TrueTime task entities. This makes it possible to use existing mutation-based testing criteria while exploiting the TrueTime ability to interact with Simulink.

Further, the paper presents a mutation-based method for generation of tests cases for testing of timeliness and control performance of flexible real-time systems. A proof-of-concept case study shows that mutation operators for testing of timeliness also can be used to produce mutants that cause control failures in flexible real-time control systems. Apart from producing test cases, the test case generation process provides a limited form of automated analysis that may increase confidence in control

system models robustness against variations in activation patterns and deviations from assumptions.

A limitation in the presented approach is that the implementation currently assumes that a specific *TAT* automata template generates the activation patterns of aperiodic tasks. The mapping function can be generalized to support a larger class of *TAT* automata templates, and thus, allow a better modeling of inherent causal dependencies between aperiodic events occurring in the environment. Future work includes investigating the scalability of the approach when generating test cases for larger and more complex control systems. In this context it is also relevant to investigate heuristics that increases the genetic algorithms ability to reveal control failures.

References

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

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.

[3] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems*. Prentice Hall, 1997.

[4] T. P. Baker. Stack-based scheduling of real-time processes. *The Journal of Real-Time Systems*, (3):67–99, 1991.

[5] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design – The ARTIST Roadmap for Research and Development*. Springer-Verlag, 2005.

[6] A. Cervin, B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Göteborg, Sweden, aug 2004.

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

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

[9] C. Houck, 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.

[10] G. Hwang, K. Tai, and T. Hunag. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.

[11] Z. Michalewicz and D. B. Fogel. *How to solve it : Modern Heuristics*. Springer, 2nd edition, 1998.

[12] F. Müeller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *In Proceedings of the 19th Real-Time Technology and Applications Symposium*, pages 179–188, Madrid, Spain, 1998.

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

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

[15] C. Nordström, A.Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Real-time Computing, Systems and Applications (RTCSA'99)*, Hong Kong, December 1999.

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

[17] A. Pettersson 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, February 2003.

[18] L. Sha, R. Rajkumar, and J. P. Lehczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 9(39):1175–1185, 1990.

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

[20] Q. Zhao, B. H. Krogh, and P. Hubbard. Generating test inputs for embedded control systems. *IEEE Control Systems Magazine*, August 2003.