Testing Deadline Misses for Real-Time Systems Using Constraint Optimization Techniques

Stefano Di Alesio[1] Arnaud Gotlieb[1] Shiva Nejati[1] Lionel Briand[1,2]

[1]Simula Research Laboratory
Oslo, Norway
{stefanod,arnaud,shiva}@simula.no

[2]SnT Center, University of Luxembourg
Luxembourg
lionel.briand@uni.lu

***Abstract*—Safety-critical real-time applications are typically subject to stringent timing constraints which are dictated by the surrounding physical environments. Specifically, tasks in these applications need to finish their execution before given deadlines, otherwise the system is deemed unsafe. It is therefore important to test real-time systems for deadline misses. In this paper, we present a strategy for testing real-time applications that aims at finding test scenarios in which deadline misses become more likely. We identify such test scenarios by searching the possible ways that a set of real-time tasks can be executed according to the scheduling policy of the operating system on which they are running. We formulate this search problem using a constraint optimization model that includes (1) a set of constraints capturing how a given set of tasks with real-time constraints are executed according to a particular scheduling policy, and (2) a cost function that estimates how likely the given tasks are to miss their deadlines. We implement our constraint optimization model in ILOG SOLVER, apply our model to several examples, and report on the performance results.**

I. INTRODUCTION

In the transport domain, safety-critical real-time applications need to be thoroughly verified before being used in operational conditions. Functional testing is usually the preferred validation technique for these applications and it usually uncovers many defects at various stages of the software development. Even if testing an application regarding to its functional behaviors can detect defects to non-functional properties (e.g., by evaluating execution time, memory consumption, etc.), computing input combinations intended to violate non-functional properties is becoming a standardized way of checking real-time applications [1], [2].

Many safety critical applications consist of concurrent tasks that are subject to real-time constraints such as deadlines. Specifically, these tasks are invoked by external events with frequent arrival times, causing each task to be executed in several execution rounds. Each task's execution must finish before a specific deadline. This must also guarantee that each task execution terminates before the next arrival of the event triggering a subsequent execution of that task. Our objective is to test real-time safety critical applications for deadline misses. We do so by finding testing scenarios that maximize the chances of deadline misses within the system. We refer to this testing activity as *stress testing* [3]. Similar to the stress testing approach of [2], we characterize the stress test scenarios, i.e., test cases, by sequences of arrival times of events triggering different executions of individual tasks.

To stress test the system, the sequences of event arrival times must be chosen such that the completion times of the tasks' executions are pushed as close as possible to their deadlines. Identifying such sequences is complicated as many tasks, with different priorities, can be triggered within a real-time application. Finding stress test cases requires to search the possible ways that a set of real-time tasks can be executed according to the scheduling policy of their underlying operating system. A classification of existing search approaches to solve such scheduling problems is difficult because of the varied nature of the software and hardware architecture model (task deadlines, priorities, platform-specific properties such as the number of processors, scheduling and memory allocation policy, etc.), the cost objective functions, and finally the solving strategies (search heuristics, complete vs incomplete solvers, etc.). Some existing approaches use constraint solvers to search for feasible task schedules (e.g., [4]–[6] use constraint solvers for *Job shop scheduling* which is a well-known optimization problem). Some other approaches rely on search-based heuristics (e.g., [2] uses genetic algorithms to generate stress test cases).

In our approach, the search for stress test cases for deadline misses is formalized using a constraint optimization model that includes (1) a set of constraints describing a declarative representation of the tasks, their timing constraints and priorities, and the platform-specific information, and (2) a cost function that estimates how likely are the given tasks to miss their deadlines. We specify the arrival times of aperiodic tasks as *scheduling variables* in our constraint optimization model. The goal is, then, to compute values for the scheduling variables such that: (a) the constraints characterizing execution of tasks on a real-time platform are satisfied, and (b) the objective function is optimized (maximized or minimized depending on the problem).

The real-time platform we are concerned with in this paper implements a *multi-processing fixed-priority preemptive scheduler*, meaning that the executions of a task can be preempted by the executions of another task if the latter has a higher priority than the former. In addition, the presence of multi-core processors allows for more than one task to be executed at any given time. We built our constraint optimization model in the Optimization Programming Language (OPL) [7]. We apply our model to several examples and report on the performance results obtained by solving the model using ILOG SOLVER [7], which is one of the leading Constraint Programming solvers on the market.

Organization: Sec. II presents a theoretical view of the problem we address in this paper. Sec. III details our OPL model for this problem. Sec. IV presents preliminary results we obtained by using ILOG SOLVER to solve our model for a number of examples. Sec. V discusses the related work, and Sec. VI concludes the paper.

II. A THEORETICAL VIEW OF MULTI-PROCESSING FIXED-PRIORITY PREEMPTIVE SCHEDULING PROBLEM

The key idea of our work is to model properties of tasks of the System Under Test (SUT) as integer variables, and to model the scheduler determining the execution of tasks as a set of constraints among those variables. Therefore, we formalize the problem addressed in this paper as a constraint optimization problem, i.e., an Integer Program (IP) over a finite domain, provided that the following assumptions are satisfied:

1) The scheduler checks the running tasks for (potential) preemptions at regular and fixed intervals of time (called *time quantum*). Each time value in our problem is expressed as an integer multiple of the time quantum.
2) The interval of time in which the scheduler switches context between tasks is negligible compared to the time quantum.

We now present a notation for the abstractions of our problem, which define variables and constants of our IP. We will not formally define all the constraints, as their definition can be easily derived by their OPL implementation reported in Sec. III.

Observation Interval. Let tq be the maximum number of time quanta we spend observing the task executions. We define $T = \{0, ..., tq - 1\}$ as the set of time quanta during which we simulate the underlying real-time application.

Tasks. Let $J = \{j_0, j_1, \dots j_{n-1}\}$ be a given set of tasks. Each task $j \in J$ has the following real-time properties:

- $exec(j)$: maximum number of executions of j within the time interval T.
- $p(j)$: priority of j.
- $dl(j)$: deadline of j, i.e., the maximum amount of time by which j should finish, after the arrival of its trigger.
- $max\_ia(j)$: maximum inter-arrival time of j, i.e., the maximum time difference between two successive arrivals of the event triggering j.
- $min\_dr(j)$ and $max\_dr(j)$: respectively minimal and maximal duration of j.

All of the above properties have fixed values which are given as the *input values* in our IP.

Task Executions. Let $A_i = \{a_{ik} \mid a_{ik}$ is the k-th execution of $j_i\}$ be a given set of executions of task j_i, and let $A = \bigcup_{i=0}^{n-1} A_i$ be the set of all the task executions. We omit the double index from task executions and say $A = \{a_0, a_1, \dots a_{m-1}\}$ when the belonging task can be inferred from the context. Each task execution $a \in A$ has the following real-time properties:

- $at(a)$: arrival time of the event triggering a
- $dr(a)$: duration of a
- $s(a)$: start time of a
- $e(a)$: end time of a
- $edl(a)$: deadline of a within the time observation interval. Note that for each a_{ik}, we have $edl(a_{ik}) = at(a_{ik}) + dl(j_i)$

All of the above properties are variable and regarded as the *output values* in our IP:

Task Ordering. Tasks are constrained to be executed in an order, implying constraints on the start and end time of the task executions. We consider two kinds of relations defining the possible orderings among task executions. Both of these relations are fixed and given, and hence, are treated as *input values* of our IP:

- Task Triggering. A task can trigger other tasks upon completion: this means that if j_1 triggers j_2, for each execution round k, we have $e(a_{1k}) = at(a_{2k})$. Obviously, the task triggering relation is asymmetric.
- Data Dependency. A task can share some computational resources with (and thus, depend on) other tasks: this means that if j_1 depends on j_2, for each execution k such that $s(a_{1k}) < s(a_{2k})$, we have $s(a_{2k}) \geq e(a_{1k})$. This is because j_1 locks the shared resource during its execution. The data dependency relation is symmetric.

Objective Function. We characterize test cases in our work by the arrival times $at(a_0) \dots at(a_{m-1})$ of task executions $a_0 \dots a_{m-1}$. To identify test cases which are more likely to lead to deadline misses, we are interested in those values for $at(a_0) \dots at(a_{m-1})$ that satisfy the constraints described in Sec. III and maximize the objective function f defined as follows:

$$f = \sum_{i=0}^{m-1} max(0, min(1, e(a_i) - edl(a_i)))$$

A set of task executions A characterizes a valid schedule if every task execution terminates before its deadline, i.e., $\forall a \in A \cdot e(a) \leq edl(a)$. The function f represents the number of deadline misses among all tasks: if a task execution a_i misses its deadline, we have $e(a_i) - edl(a_i) > 0$, and thus a_i contributes for 1 to the value of f. Otherwise, we have $e(a_i) - edl(a_i) \leq 0$, and thus a_i contributes for 0 to the value of f.

III. OUR OPL MODEL

ILOG CPLEX [7] is a widely known Constraint Programming environment to address various constraint satisfaction and optimization problems. In our work, we particularly relied on using Finite Domain (FD) constraint solving capabilities of ILOG SOLVER. In this section, we describe the OPL model we designed to solve test case generation for the dead-line miss problem. As some of the variables and constraints are trivially defined from their formulation, we intentionally do not describe all of them, focusing on the description of novel structures and complex constraints. The full implementation of the OPL model can be found at [8].

A. Constants

Constants in our model have their values assigned by external data, as it is common practice in the OPL language. Each

constant value, as the number of time quanta tq, the number of tasks n, and the number of cores c is implemented as a constant integer. Each quantity relative to a task described in Sec II is implemented as a constant integer array ranging over the set J of tasks. Task Triggering and Data Dependency ordering relations are defined as boolean matrices named $triggers$ and $dependent$. The $triggers$ and $dependent$ matrices are defined as follows:

$$triggers(j_i, j_k) = \begin{cases} 1 & \text{if } j_i \text{ triggers } j_k \\ 0 & \text{otherwise} \end{cases}$$

$$dependent(j_i, j_k) = \begin{cases} 1 & \text{if } j_i \text{ depends on } j_k \\ 0 & \text{otherwise} \end{cases}$$

Listing 1. Constants

```
// T: Observation interval (range of time quanta)
int tq = ...;
range T = 0..tq-1;

// c: Number of Processor Cores
int c = ...;

// n: Number of tasks
int n = ...;
range J = 0..n-1;

/* Task Constants */
int priority[J] = ...;
int task_deadline[J] = ...;
int max_interarrival_time[J] = ...;
int min_duration[J] = ...;
int max_duration[J] = ...;
int triggers[J, J] = ...;
int dependent[J, J] = ...;

/* Task Execution Constants */
int task_executions[J] = ...;
{TaskExecution} A =
    {<j,k> | j in J, k in 0..task_executions[j]-1};

int est[a in A] = 0;
int lst[a in A] = tq;
int eet[a in A] = 0;
int let[a in A] = tq;
```

B. Variables

The real-time values related to task executions described in Sec II are implemented using variable integer arrays ranging over the set A of task executions. Some variables being defined by equality constraints, like $dr(a)$ and $edl(a)$ were defined as variable expressions in order to increase performance. This is a standard practice in OPL [7]. We also defined two additional variables to simplify the description of our constraints:

- Active. As known in the literature [5], preemptive scheduling problems can be encoded using a time-table data structure. Thus, we defined a similar structure named $active$ as a boolean matrix with m rows, each one corresponding to a task execution $a \in A$, and tq columns, each one corresponding to a time quantum. The $active$ matrix is defined for each task execution a and each time quantum $t < tq$ as follows:

$$active(a, t) = \begin{cases} 1 & \text{if } a \text{ is executing at time } t \\ 0 & \text{otherwise} \end{cases}$$

- Eligible for Execution. To be able to describe the preemption constraints in our model, we introduced a variable named $eligible\_for\_execution$, implemented as an array ranging over the set A of task executions. Specifically, $eligible\_for\_execution(a)$ represents the time when task execution a could start assuming an unlimited number of cores are available, i.e. when the maximum degree of parallelization is allowed. The time when a task execution a_{ik} is said to be *eligible for execution* is the latest time between its arrival time $at(a_{ik})$ and the end time of its previous execution $e(a_{ik-1})$. We are unaware of any other related work in task allocation scheduling defining a similar structure.

Each variable is also defined with a range constraint, capturing its associated finite domain. As known in the literature [5], [6], bounds for start and end times of task executions are referred as *earliest/latest start/end time*.

Listing 2. Variables

```
/* Task Execution Variables */
dvar int arrival_time[a in A] in T;
dvar int eligible_for_execution[a in A] in est[a]..lst[a];
dvar int start[a in A] in est[a]..lst[a];
dvar int end[a in A] in eet[a]..let[a];
dvar int active[a in A, t in T] in 0..1;
dexpr int duration[a in A] = sum(t in T) active[a, t];
dexpr int task_execution_deadline[a in A] =
    minl(arrival_time[a] + task_deadline[a.task], tq);
dexpr int deadline_miss[a in A] =
end[a] - task_execution_deadline[a];
```

C. Well-Formedness Constraints

In addition to range constraints, we add a set of *Well Formedness* constraints to our model to capture relations among the variables which directly follow from the definitions given in Section II. These constraints may specify the valid range-values for variables such as the constraints used to define the $active$ matrix (labeled as *wf9-wf12*), or they may describe how one variable is bounded by other variables such as the constraints used to define $s(a)$ for a task execution a (labeled as *wf3-wf4*).

Listing 3. Well-Formedness constraints

```
/* I. Well-formedness constraints */
forall(a in A) {

    if(prevc(A, a).task == a.task &&
        sum(al in A) triggers[al.task, a.task] == 0) {
        wf1: arrival_time[prevc(A, a)] +
            task_deadline[prevc(A, a).task] <=
                arrival_time[a];
        wf2: arrival_time[a] <= arrival_time[prevc(A, a)] +
            max_interarrival_time[prevc(A, a).task];
    }

    wf3: eligible_for_execution[a] <= start[a];
    wf4: start[a] <= end[a];

    if(prevc(A, a).task == a.task)
        wf5: eligible_for_execution[a] ==
            maxl(arrival_time[a], end[prevc(A, a)]);
    else
        wf6: eligible_for_execution[a] == arrival_time[a];

    wf7: min_duration[a.task] <= duration[a];
    wf8: duration[a] <= max_duration[a.task];

    forall(t in T) {
        wf9: t == start[a] => active[a, t] == 1;
        wf10: t == end[a] - 1 => active[a, t] == 1;
```

```
        wf11: t <= start[a] - 1 => active[a, t] == 0;
        wf12: t >= end[a] => active[a, t] == 0;
    }
}
```

D. Temporal Ordering Constraints

Temporal Ordering constraints capture the task triggering (constraint *to2*) and data dependence (constraint *to3-to4*) relations defined in Sec. II. For instance, if task j_1 triggers the execution of task j_2, the arrival time of each task execution a_{2k} is equal to the end time of each task execution a_{2k}. In addition, the constraint *to1* ensures that the start time of each task execution follows the end time its predecessor.

Listing 4. Temporal Ordering constraints

```
/* II. Temporal Ordering constraints */
forall(a in A) {

    forall(a1 in A : a1.task == a.task &&
           a1.execution == a.execution-1)
        to1: start[a] >= end[a1];

    forall(a1 in A : triggers[a.task, a1.task] == 1)
        to2: end[a] == arrival_time[a1];

    forall(a1 in A : dependent[a.task, a1.task] == 1) {
        to3: start[a] != start[a1];
        to4: start[a] <= start[a1]-1 => start[a1] >= end[a];
    }
}
```

E. Multicore Constraint

The multicore constraint ensures that for each time quantum t the number of executing tasks does not exceed the number c of available cores. This is done by summing up, for each t, the values of $active(a, t)$ of all the task executions a.

Listing 5. Multicore constraint

```
/* III. Multi-core Constraint */
forall(t in T)
    mc: sum(a in A) active[a, t] <= c;
```

F. Preemptive Scheduling Constraints

As we said in Sec. I, the scheduler modelled by our constraint program has a priority-based preemptive policy. The Preemptive Scheduling constraints define such behavior specifying that each task can potentially be preempted during each execution (constraint *ps1*), and that if at a given time quantum t task execution a_1 is running and task execution a_0 is not because the platform cores are all busy, then a_1 has an higher priority than a_0 (constraint *ps2*). Due to lack of coupling between the constraints sets, our model can be tailored to various applications. For example, this set of constraints can be replaced by another one defining a different scheduling policy, e.g., Round Robin. This allows the adaptation of our solution to various applications.

Listing 6. Preemptive Scheduling constraints

```
/* IV. Preemptive Scheduling Constraints */
forall(a in A)
    ps1: end[a] - start[a] >= duration[a];

forall(t in T, a0 in A, a1 in A)
    ps2: (active[a0, t] == 0 &&
          active[a1, t] == 1 &&
          sum(a2 in A) active[a2, t] == c &&
          eligible_for_execution[a0] <= t &&
          end[a0] >= t+1)
              =>
          (priority[a1.task] >= priority[a0.task]);
```

G. Good CPU Usage Constraints

This set of constraints ensure that the scheduler avoids unnecessary context-switching between task executions, and optimizes its resource usage by running tasks which are eligible for execution as soon as there are free cores. Due to lack of space, these constraints are not described here, but are available at [8].

IV. Experiences

In this section, we provide some initial performance results by applying ILOG Solver to solve our OPL model for a number of examples. When conducting this study, we had two main goals. The first goal was to demonstrate correctness of our constraint model, and our second goal was to evaluate the performance of our technique when the observation time and the number of tasks in the system change.

Correctness. Let us consider a set J of three tasks $j0$, $j1$ and $j2$, with properties shown in Table I, running during an interval of $tq = 10$ time quanta on a scheduler with $c = 2$ cores.

	Task $j0$	Task $j1$	Task $j2$
$exec(j)$	2	2	2
$p(j)$	100	101	102
$dl(j)$	3	2	3
$max\_ia(j)$	3	2	3
$min\_dr(j) = max\_dr(j)$	3	2	3

Table I
Real-time properties of our example tasks.

The result of our OPL model shows that there are 20 possible assignments for arrival times of tasks $j0$, $j1$, and $j2$ which maximize the objective function f described in Sec. II, one of which is shown in Lst. 7.

Listing 7. One of the solutions for our OPL model

```
arrival_time (at): [0 3 2 4 0 3]
duration (dr): [3 3 2 2 3 3]
eligible_for_execution (efe): [0 7 2 4 0 3]
start (s): [0 7 2 4 0 3]
end (e): [7 10 4 6 3 6]
task_execution_deadline (edl): [3 6 4 6 3 6]
deadline_miss: [4 4 0 0 0 0]
active: [[1 1 0 0 0 0 1 0 0 0]
         [0 0 0 0 0 0 0 1 1 1]
         [0 0 1 1 0 0 0 0 0 0]
         [0 0 0 0 1 1 0 0 0 0]
         [1 1 1 0 0 0 0 0 0 0]
         [0 0 0 1 1 1 0 0 0 0]]
```

Since each task in our example in Table I is executed twice, this example includes six task executions i.e., $a_{01}, a_{02}, a_{11} \ldots a_{22}$. The rows of the $active$ matrix correspond to the task executions (6 task executions), and its columns correspond to the time quanta ($tq = 10$). Lst. 8 shows the active matrix in Lst. 7 annotated with the row and column labels. The scenario that this matrix shows leads to a deadline miss for task $j0$. At $t = 2$, task j_0 has to be preempted because j_2 and j_1 have higher priorities and occupy the two existing

cores. This task can resume its first round of execution, a_{01}, only at $t = 6$, thus missing its deadline at $t = 4$.

Listing 8. The *active* matrix for the solution in Lst. 7

```
1  t = 0 1 2 3 4 5 6 7 8 9
2 a01 [1 1 0 0 0 0 1 0 0 0]  1st round j0
3 a02 [0 0 0 0 0 0 0 1 1 1]  2nd round j0
4 a11 [0 0 1 1 0 0 0 0 0 0]  1st round j1
5 a12 [0 0 0 0 1 1 0 0 0 0]  2nd round j1
6 a21 [1 1 1 0 0 0 0 0 0 0]  1st round j2
7 a22 [0 0 0 1 1 1 0 0 0 0]  2nd round j2
```

Performance. In order to analyze the performance of our approach, we created different versions of the example in Table I by varying the number of tasks, i.e., n. We applied our OPL model to these different versions and observed the results by changing the time interval (i.e., the number tq of time quanta). In our study, we used the CPLEX solver of ILOG. For each case, we let the solver run for a maximum of 3 hours on a MacBook Pro geared by a 2.0 GHz Intel Core i7 with 8GB RAM. In cases where the CPLEX solver did not terminate in less than 3 hours, we recorded the solving time only for finding the first optimal solution. Otherwise, we also recorded the time for finding each individual solution. The results of our study are shown in (Fig. 1).

Our results (summarized in the three top-graphs and the bottom-left and bottom-middle graphs) in Fig. 1 show that increasing tq and n both slow down the solving time, but tq results in a steeper slow-down than n. The slow-down is higher when we consider the time required to find all the solutions, and lower when we consider that for one solution. In addition, as shown in the bottom-right graph in Fig. 1, CPLEX can generally find most optimal solutions shortly after the search starts. However, it often takes a great deal more time for the search to terminate and to conclude that no more optimal solutions exist.

V. Discussion and Related Work

In this section, we discuss the advantages and drawbacks of our approach compared to the alternative techniques that have been previously used for analysis of real-time systems. The techniques for analysis of real-time systems are divided into two general groups: (1) Approaches based on real-time scheduling theory [9]. These approaches estimate schedulability of a set of tasks through customized formulas and theorems that often assume worst case situations only such as worst case execution times, worst case response times, etc. These approaches are often too conservative because due to inaccuracies in estimating worst-case time values, the worst-case situations may never be realized in practice. Therefore, in general, we cannot rely on schedulability theory alone when dealing with analysis of real-time systems. Moreover, extending these theories to multi-core processors has shown to be a challenge [10], [11]. (2) Model-based approaches to schedulability analysis. The idea of this approach is to base the schedulability analysis on a system model that captures the details and specifics of real-time tasks. This provides the flexibility to incorporate specific domain assumptions and a range of possible scenarios, not just the worst cases [2], [12], [13]. Furthermore, approaches that fall in this category can deal with multi-core processors as well [12], [13].

Our work falls into the second group where we create a constraint model of real-time concurrent applications and use a constraint optimization tool, i.e., ILOG Solver, to analyse the model. The main difference between our work and other existing model-based approaches resides in the choice of the analysis method. Here, we discuss and compare with our work, i.e., constraint programming (**cp**), two other techniques that have been widely used for analysis of real-time concurrent models: (1) Model checking (**mc**), in particular, real-time model checkers, e.g., UPPAAL [14], and (2) Meta-heuristic search, and in particular, Genetic algorithms (**ga**) [2].

All approaches to performance engineering and schedulability analysis require a model of the time and concurrency aspects of the system under analysis. The input models for **mc** are often a set of parallel state machines augmented with real-time information. To analyze deadline misses using **mc** approaches, the state machines should include some designated error states chosen in a such a way that the paths leading to these states represent some deadline-miss scenarios. This allows to reduce checking deadline misses to checking reachability of error states in the composition of a set of parallel state machines. In theory, model checkers are sound and complete in the sense that they are able to find an error if it exists, and when they terminate successfully without finding any error, the model is conclusively error-free. However, in practice, model checkers may fail to return any conclusive results when the state machine composition becomes too large to handle.

The **mc** approaches have been mainly used for *verification*, i.e., to check if a given set of real-time tasks satisfy some property of interest. To adapt **mc** approaches to checking different properties of real-time applications, the underlying state machines may need to be modified so that the question at hand can be formulated as a reachability query. For example, in [13], in order to analyze CPU-time usage, an *idle* state-machine was added to the set of interacting timed-automata to keep track of the CPU-time, and the error states were chosen so that their reachability could lead to violation of the CPU-time usage limit.

Declarative constraints used in our work, **cp**, and chromosome encodings required in the **ga** approaches can potentially be extracted from many existing concurrent behavioral design models containing real-time information. In particular, in [2], [12], it is shown that the constraints and the chromosome encodings can be extracted from standard UML/MARTE models [15]. In **cp** and **ga** approaches, the property to be checked is captured by a *quantitative* objective function as opposed to a *boolean* reachability property, as in the case of **mc**. Therefore, **ga** and **cp** approaches are more geared towards optimization with applications in test-case generation rather than verification. To adapt these approaches to check other kinds of real-time properties, it often suffices to change the objective function. For example, in order to change our approach to perform CPU usage analysis, we only need to modify the objective function [12].

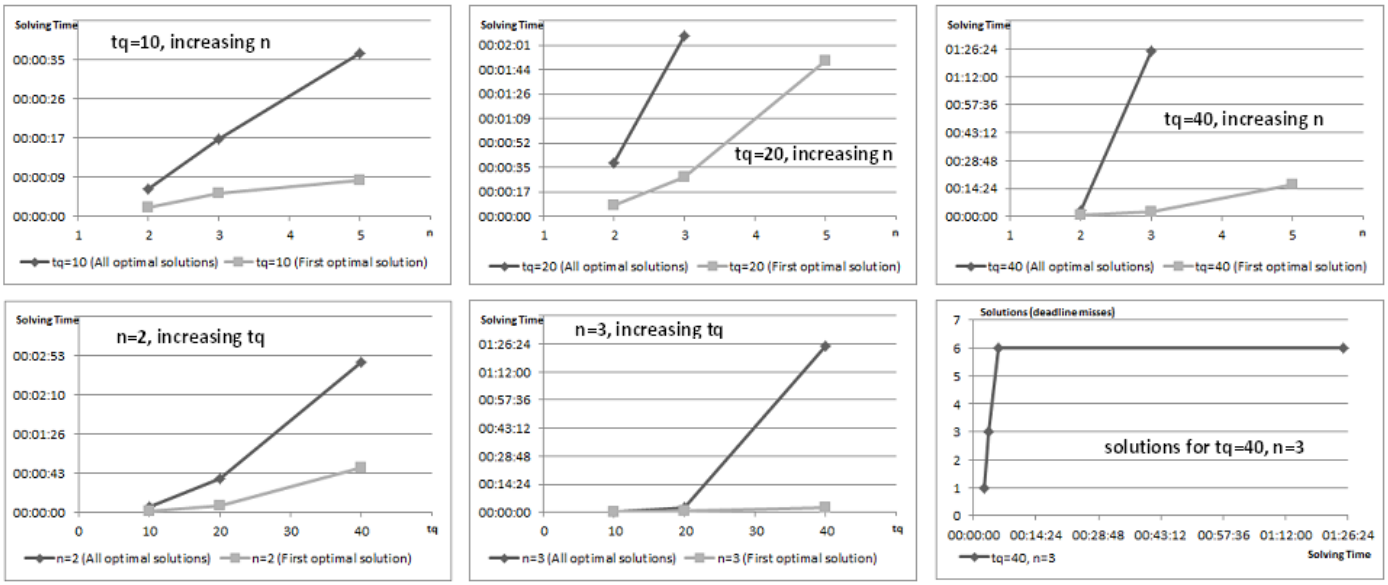

Figure 1. The performance results of our OPL model with CPLEX

Our approach, **cp**, differs with the **ga** approach in two main respects: (1) We use the complete search method of ILOG SOLVER in our work, whereas the **ga** search approach is incomplete. In other words, in our work, if the search terminates within the time allotted, the result is guaranteed to be a global optimal, while in **ga**, we can never know whether the search result is a local or a global optimal. (2) In **cp**, the choice of the objective function might be limited by the choice of the solver used for analysis of the constraints. For example, we cannot use non-integer objective functions with finite-domain linear solvers, while this limitation does not exist in **ga** implementations. In the end, we note that a precise and thorough comparison of the **mc**, **ga** and **cp** approaches, and in particular their scalability, requires extensive, large, and systematic case studies and experiments, and is not within the scope of this paper.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we provided an approach for generating test cases that can stress a real-time system to miss its deadlines. Our solution relies on existing constraint optimization techniques. We described our constraint model in OPL, and used ILOG CPLEX solver to evaluate our model on a number of examples. The correctness testing showed that, in general cases, this approach can effectively lead to the generation of test cases which are more likely to lead to deadline misses. We envision that this approach has the potential to be successfully applied to industrial contexts, providing that the efficiency of the solving computation is improved. Thus, we plan to replace the structures used in our OPL model, in particular the *active* matrix, with more efficient alternatives. Currently, the size of this matrix grows proportionally with the size of time quanta and the number of task executions grow. We plan to implement this matrix using an array of sequences of intervals of time quanta instead. Improving the matrix structure can significantly reduce the search space of our problem, leading to much shorter solving times.

REFERENCES

[1] N. R. Council, D. Jackson, and M. Thomas, *Software for Dependable Systems: Sufficient Evidence?*, 2007.

[2] L. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006.

[3] B. Beizer, *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990.

[4] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *INFORMS Journal on Computing*, vol. 3, no. 2, pp. 149–156, 1991.

[5] C. L. Pape and P. Baptiste, "Resource constraints for preemptive job-shop scheduling," *Constraints*, vol. 3, no. 4, pp. 263–287, 1998.

[6] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien, "Solving a real-time allocation problem with constraint programming," *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, 2008.

[7] "IBM ILOG CPLEX Optimization Studio Information Center," http://publib.boulder.ibm.com/infocenter/cosinfoc/v12r3/index.jsp.

[8] S. Di Alesio, "The deadline-miss constraints in ILOG SOLVER," http://home.simula.no/∼stefanod/ilog.pdf.

[9] J. W.-S. Liu, *Real-time systems*. Prentice Hall, 2000.

[10] M. Bertogna, "Real-time scheduling analysis for multiprocessor platforms," Ph.D. dissertation, Scuola Superiore Sant'Annna, Pisa, 2007.

[11] A. David, J. Illum, K. Larsen, and A. Skou, *Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1*. CRC Press, 2010, pp. 93–119. [Online]. Available: http://www.cs.ru.nl/∼fvaan/PC/chapter.pdf

[12] S. Nejati, S. Di Alesio, M. Sabetzadeh, and L. Briand, "Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing," Simula Research Laboratory, Tech. Rep. 2010-18, 2011.

[13] M. Mikucionis, K. Larsen, B. Nielsen, J. Illum, A. Skou, S. Palm, J. Pedersen, and P. Hougaard, "Schedulability analysis using UPPAAL: Herscehl-Planck case study," in *ISoLA*, 2010.

[14] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," in *SFM-RT 2004. Revised Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer Verlag, 2004, pp. 200–237.

[15] "A UML profile for MARTE: Modeling and analysis of real-time embedded systems," May 2009.