
Gerhard Fohler
Department of Computer Engineering
Malardalens University
S-72123 Vasteras, Sweden
gerhard.fohler@mdh.se

1 Introduction and Motivation

The application of standard timing constraints, such as deadlines and periods, is facing two limitations: They can overconstrain specifications and lack expressive power.

Only few tasks have "natural" periods and deadlines. Most are only derived during system design: they are artifacts [Sta90]. Often, several settings for deadline and period may be feasible, but only one can be selected, thus abandoning the knowledge of more flexibility: the specification becomes overconstrained. When the scheduler fails on that timing constraints, a complete re-design and re-scheduling has to be performed.

Many real-world applications, e.g., control applications, have timing requirements which cannot be expressed with deadlines and periods or which overconstrain the system strongly if expressed by standard timing constraints.

Gerber et. al. have studied relative timing constraints [Ger95b] and design based on end-to-end deadlines [Ger95a]. Cheng and Agrawala [Che97] developed a scheduling algorithm for relative timing constraints. Wirz and Schild [Wur97] proposed using constraints satisfaction methods to schedule relative timing constraints. [Foh94] presented a design method integrating RTL [Jah86] and precedence constraints, and a static scheduler to fulfill the combined requirements. The slot shifting method for static schedules [Foh95] is capable of supporting limited dynamic timing constraints.

In this paper, we present dynamic timing constraints, which help to overcome the above mentioned problems. We introduce timing entities, representing functional units. These are given a feasibility function describing the temporal requirements instead of providing concrete constraints such as period and deadline. An optional instantiation function creates instances of feasible timing constraints upon request.

A specification with dynamic timing constraints supports several types of interactions with scheduling: Selecting appropriate standard timing constraints, changing timing constraints during schedule construction, and scheduling timing entities directly, avoiding artifacts of deadlines and periods altogether.

The rest of this paper is organized as follows: Section 2 starts by discussing limitations of standard timing constraints. A presentation of dynamic timing constraints follows in section 3. Scheduling issues are addressed in section 4. Summary and outlook in section 5 conclude the paper.

2 Limitations of Standard Timing Constraints

In this section, we describe limitations of standard timing constraints and discuss their impact using sporadic event handling as example.

2.1 Pseudo-periodic Tasks - Example

Let us have a look at the steps a designer takes to transform his knowledge about the timing of handling sporadic events into the concrete timing constraints period and deadline.

Designer's knowledge: Upon occurrence of a sporadic event e, the system has to react within a given time interval, react. While the exact occurrence times of e cannot be determined beforehand, a minimum interarrival time, mint, can be given. A task, Te, with maximum execution time c(Te) is provided to handle e.

Task model requirements: Due to system requirements, e has to be handled by a periodic task, i.e., period and deadline have to be set for Te.

Temporal feasibility: The instances of Te handle e temporally feasible, when 1) each occurrence of e is handled by an instance of Te within react and 2) no occurrence of e is missed.

1. In the worst case, an instance e1 occurs - at time t(e1) - immediately after an instance T2 has begun execution, checked for e and found it had not occurred. e1 will be handled by the next instance, Te+1.

   dl(Te+1) - t(e1) < react. dl denotes deadline.
2. Two instances of e are separated by at least \( \text{mint} \).
   In order to not miss any, two consecutive instances of \( T_e \) have to start less than \( \text{mint} \) apart.
   \( st(T_e^{i+1}) - st(T_e^i) < \text{mint} \). \( st \) denotes start time.

Note that these feasibility conditions allow different timing constraints for individual instances of \( T_e \).

**Feasible timing constraints imposed by task model:** As the task model required by the system demands period, \( p \), and deadline, \( dl \), as timing constraints, the designer has to transform the feasibility conditions accordingly.

1. \( p(T_e) + dl(T_e) < \text{react} \)
2. \( p < \text{mint} \)

Note that various pairs for deadline and period can possibly meet these requirements.

**Selection of single period, deadline pair:** In the last step, the designer has to select a single pair for period and deadline values, thus abandoning other equally feasibly pairs.

To illustrate this selection, let us assume the following data for e and \( T_e \): \( \text{react} = 61 \), \( \text{mint} = 70 \), \( c(T_e) = 9 \).

From the set of feasible pairs of period, deadline we select five, \( (50, 10), (40, 20), (30, 30), (20, 40) \), and \( (10, 50) \) for further study.

The system consists of \( T_e \) and two periodic tasks, \( T_p \): \( c(T_{p_0}) = 10 \), \( p(T_{p_0}) = 30 \), \( dl(T_{p_0}) = 30 \), and \( T_{p_1} \): \( c(T_{p_1}) = 30 \), \( p(T_{p_1}) = 70 \), \( dl(T_{p_1}) = 70 \). We now calculate the utilization of the system.

<table>
<thead>
<tr>
<th>( T_{p_0} )</th>
<th>( T_{p_1} )</th>
<th>( T_e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>dl</td>
<td>15</td>
<td>70</td>
</tr>
<tr>
<td>( p )</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>( u_{\text{total}} )</td>
<td>0.94</td>
<td>0.99</td>
</tr>
</tbody>
</table>

The pair \( (10, 50) \) has the lowest utilization demand, but also the tightest deadline, making it harder to find feasible schedules. In fact, assuming all tasks first start at time 0, there will be a collision between \( T_{p_0} \) and \( T_e \) at time 150, which cannot be averted, due to \( dl(T_e) \) allowing no shift of \( T_e \).

This problem can be circumvented by selecting a more relaxed value for \( dl(T_e) \). As we can see in the table, however, the new taskset may become infeasible due to increased processing demands of \( T_e \).

Note that this feedback is provided by schedulability analysis and only available after the entire system has been specified.

**Conclusion:** We can see that the specification becomes increasingly overconstrained. With the given timing constraints, the information about possible options and other selection is lost in each step. Instead of representing the information about the temporal feasibility of the task, the result in the form of numbers is kept only.

### 2.2 Quality of Control Algorithm

The quality requirements of a control algorithm translate into requirements on the timing of the controlling tasks, e.g., sampling period and feedback loop delays with tolerances \( [\text{Tor}] \). The designer is faced with selecting concrete values for, e.g., period and deadline. What is a desired setting may turn out to be unschedulable. The chosen values may, however, be changed, as long as they are within the given tolerances, thus keeping the quality of the control algorithm.

### 2.3 Further Limitations

We list further examples for limitations of standard timing constraints. **Data dependent requirements**, in which the timing properties, e.g., periods depend on input data, e.g., reading sensor data; **time dependent requirements**, where the actual timing, e.g., the deadline becomes known only at run-time, e.g., setting actuators; **quality of service requirements**, which demand constraints involving several instances, e.g., "in out of n"; **decomposition of end-to-end deadlines and distributions**, which transform into start times and deadlines.

### 3 Dynamic Timing Constraints

We propose to keep information about the temporal feasibility and alternative values encountered during the design process, as opposed to the resulting numbers only, in so-called timing entities for fast access during design and scheduling.

**Timing entity (TE):** A TE is a functional unit, i.e., a time constrained system activity, such as a task or a message, combined with a feasibility function, which tests the current timing of the TE. Optionally, an instantiation function can be provided for generation of feasible timing settings.

Most activities in a real-time system are periodic and will have a number of instances during operation. Therefore, we envision the following types of constraints.

**Instance timing constraints (ITC):** The set of timing constraints for a single instance of a TE is called instance timing constraints.

**Static TC (sTC):** TCs of a TE which remain fixed for the entire run-time are called static timing constraints. The instance timing constraints are the same for all instances of that TE.

The standard TCs period, deadline, etc are typically sTC.

**Dynamic TC (dTC):** TCs of a TE which may vary for each instance of a TE, are called dynamic timing constraints. The instance timing constraints may be different for each instance.
Consider the example of section 2, the timing of a periodic task to handle a sporadic event was derived (static timing constraints). There was a trade-off between collision due to stringent deadline and high processing demands. Using dynamic timing constraints solves the problem by relaxing the deadline (and raising the load) of the “collision” instance only, changing its value back for the next instance. The handling of the sporadic event would still be feasible, the collision avoided, and load kept acceptable.

**Feasibility Function (FF):** Given a set of static or instance TC, for a TE, the feasibility function determines whether the temporal requirements on the TE are met by these TC. **Instantiation Function (IF):** The instantiation function of a TE instantiates a set of feasible static or instance ITC. Repetitive application of IF produces different sets of ITC. Apart from testing whether a set of TC is feasible, it may also be useful to be able to generate - and different - feasible TCs. If a TC setting is abandoned by the designer or the scheduler, the TE can provide a new setting. We are currently investigating into the “smart” selection of new instances, since it might benefit design and scheduling in an “heuristic” way.

**Example - Pseudo periodic task:** We illustrate the construction of TEs with the example of section 2.

**Functional unit:** The task $T_e$ handling the event $e$.

**Dynamic instance timing constraints** of instance $i$ are determined from the temporal feasibility information of the sporadic event:

- $i = 0: \text{starttime}(T^i_e) < \text{mint}$
- $i > 0: \text{starttime}(T^i_e) - \text{starttime}(T^{i-1}_e) < \text{mint}$
- $\text{endtime}(T^i_e) - \text{starttime}(T^{i-1}_e) < \text{react}$

An instantiation function could vary starttimes and endtimes.

**Static instance timing constraints** $p(T_e) + dl(T_e) < \text{react}, p < \text{mint}$ An instantiation function could vary $p$ or $dl$.

**4 Scheduling Issues**

We will present three approaches to scheduling dTCs with increasing utilization of dTCs.

**4.1 Selecting Static Timing Constraints**

If a standard scheduling algorithm working on standard timing constraints is mandated, a dTC specification is still beneficial. Whenever an overconstraining artifact is introduced in the design process, we also know about different options and their feasibility. After specification we try to construct a schedule or perform a schedulability test. If this fails, we have already pinpointed standard constraints candidates for instantiation and re-submission.

**4.2 Selecting Dynamic Timing Constraints**

A static scheduler can use dynamic timing constraints during schedule construction. Although these remain fixed once the schedule is constructed, the scheduler has more flexibility in constructing the schedule and more solutions can be found. Another improvement over standard timing constraints is that the scheduler can consider, select, and undo various individual constraints expressed by dTCs during schedule construction and not only for all instances after a complete run.

**4.3 Scheduling Timing Entities**

The previous methods have proposed to use modified standard scheduling methods and standard timing constraints as a baseline for the application of dynamic timing constraints: The higher level information of the timing entities is used to instantiate lower level constraints used by the algorithms. These constraints are possibly abandoned and replaced by new ones provided by the information in the timing entities. The “trial-and-error” instantiation of constraints becomes a bottleneck for the sake of standard scheduling algorithm requirements.

We are therefore investigating eliminating the instantiation step altogether and develop an algorithm that is capable of scheduling timing entities directly. Currently, we are studying constraint satisfaction programming methods for their applicability. At this point, we feel they are well suited due to their support for constraint specification and solution algorithms.

**5 Summary and Outlook**

In this paper, we have presented a novel form for the representation of information about the temporal correctness of activities in real-time systems, called dynamic timing constraints.

We discussed limitations of standard timing constraints and how the design process overconstrains specifications by introducing artifacts and abandoning alternatives during their derivation. Dynamic timing constraints represent conditions for the temporal correctness rather than fixed values for constraints such as period and deadline. This is achieved by so-called timing entities, which combine a functional unit, such as a task, with a feasibility function for testing the feasibility of the timing of the unit. Optionally, an instantiation function can be given to generate various alternative timing constraints. This representation allows the system specification to provide information about feasibility and various options of time related design decisions. The designer or scheduler can use this information to select timing constraints that seem best suited. Misleading choices can be identified and replaced by other options immediately avoiding major consideration during redesign.

We have outlined how dynamic timing constraints can be used with standard scheduling algorithms, indicated
modifications to these algorithms, and novel approaches fully utilizing the benefits of dynamic timing constraints.

Our current research in the area includes the following issues: Timing entities are designed to keep information about single system activities. Questions are if a hierarchical composition of timing entities would be useful for earlier stages of the design process as well and whether modularization can and should be aimed at. We are in the process of modifying the scheduler described in [Foh94] to instantiate and change timing constraints during schedule construction. Further, we are developing an algorithm based on constraint satisfaction programming to schedule timing entities directly.

Acknowledgements
The author gratefully acknowledges the fruitful discussions, stimulating suggestions, and encouraging comments by Krithi Ramamritham.

References


