`WeaklyHard.jl`: Scalable Analysis of Weakly-Hard Constraints

Nils Vreman[1], Richard Pates[1], Martina Maggio[2,1]
[1]Department of Automatic Control, Lund University, Sweden
[2]Department of Computer Science, Saarland University, Germany

Abstract—**Weakly-hard models have been used to analyse real-time systems subject to patterns of deadline hits and misses. However, the tools that are available in the literature have a set of shortcomings. The analysis they offer is limited to a single weakly-hard constraint and to patterns that specify the number of misses, rather than the number of hits. Furthermore, the scalability of the tools is limited, effectively making it hard to address systems where deadline misses are really sporadic events. In this paper we present `WeaklyHard.jl`, a scalable tool to analyse a set of weakly hard constraints belonging to all the four types of weakly hard models. To achieve scalability, we exploit novel dominance relations between weakly-hard constraints, based on deadline hits. We provide experimental evidence of the tool's scalability, compared to the state-of-the-art for a single constraint, a thorough investigation of hit-based weakly-hard constraints, and a sensitivity analysis to constraint set parameters.**

I. Introduction

A recent survey on the state of industrial practice in real-time systems showed that a significant fraction of real-time tasks are allowed to miss a finite number of deadlines [2]. The research community spent years defining and analysing models of tasks that can miss deadlines, from soft real-time systems [8], to tasks with a skip-factor [19], from calculating the miss ratio based on execution time probability distributions [23], to approximating the deadline miss probability [32] for a given system.

One of such models in which tasks may miss deadlines is the weakly-hard task model [5]. Weakly-hard tasks behave according to patterns of hit and missed deadlines that are (mainly) window-based. The originally proposed constraint models specifies alternatively (for a window of subsequent jobs): (i) the minimum number of deadlines that are hit, (ii) the minimum number of consecutive deadlines that are hit, (iii) the maximum number of deadlines that may be missed, or (iv) the maximum number of consecutive deadlines that may be missed. The third of these models – often called the (m, K) model – gained attention in the research community, generating results on scheduling algorithms [12], real-time and schedulability analysis [13], [25], [28], verification [3], [17] and runtime monitoring [34] of constraint satisfaction, derivation of task model parameters [35], together with applications to domains like telecommunication [1], [18] and control systems [24], [26]. The fourth model has also proved relevant to perform analyses of the stability of control systems [22]. Furthermore, the relation between weakly-hard constraints has been partially investigated [29], [34].

The practical usefulness of weakly-hard models will remain limited, unless it is possible to build tools to enforce and monitor the satisfaction of weakly-hard constraints for execution platforms. Many real-time platforms offer the possibility to invoke "protected" task executions, ensuring that deadlines are met at the cost of increasing the execution cost. This is a very simple mechanism to secure that the weakly-hard constraint is satisfied in an execution platform. However, this requires writing monitoring code, that generates transition points to this protected execution mode when a constraint might otherwise be violated. Generating this code in a scalable way requires abstracting from the constraint and representing the execution of tasks with compact, but expressive, models.

To date, the literature has focused on the (m, K) constraint, neglecting the others, despite their relevance in application domains such as control [20], [22]. As a result, the mentioned tools and models are not available for all the constraint types. This paper aims at both solving this problem and answering some open issues, namely: (i) guaranteeing consecutive deadline hits, and not only following patterns of deadline misses; and (ii) dealing with systems that satisfy multiple weakly-hard constraint simultaneously.

The first issue comes from the consideration that in practice it may be easier to guarantee that some prescribed job will hit their deadline rather than ensuring that the number of misses follows a given pattern. This is the case of the mentioned protected execution environment. As an example, mixed-criticality allows the scheduler to raise the criticality level and thus guarantee that the highly-critical tasks meet the corresponding deadlines [7]. We can treat the weakly-hard task as highly critical and raise the criticality level when a deadline hit must be enforced. Alternatively, we can increase the budget of a reservation-based scheduler [9]. Despite the fact that guaranteeing hits is often easier than enforcing miss patterns, the first two types of weakly hard tasks, that constrain the number of hits, have not been receiving much attention from the research community.

Furthermore, we would like to analyse tasks that satisfy multiple constraints simultaneously. Most analysis methods only take into account a single constraint, e.g., [24] or [22] for the stability of control systems. In some cases, one of the two constraints *dominates* the other, meaning that satisfying the dominant constraint also guarantees the satisfaction of the dominated one. But this is not always the case. Consider for example two constraints λ_1 and λ_2, where λ_1 specifies that the task may miss a maximum of 2 deadlines in every window of 5 consecutive jobs, and λ_2 that it may miss a maximum of 3 deadlines in every window of 7 consecutive jobs. On the one hand the sequence 0011100, where 0 represents a deadline miss and 1, satisfies λ_1 but fails λ_2, meaning that λ_2 does not dominate λ_1. On the other hand the sequence 0001111 satisfies λ_2 but fails λ_1, and so λ_1 does not dominate λ_2 either. If the analysis can only be conducted with a single constraint, the choice of which constraint is to be used is left to the practitioner, while it would be best to consider *both* constraints simultaneously.

Finally, we bring forward the question of *scalability*. Many of the research results, for example in the control domain [20], [21], [24], use short windows. However, for practical applications it may be relevant to use a large window size, as done for example in the experimental analysis in [3]. In fact, the original motivation behind the weakly-hard task model [5] uses a practical example from the avionics domain in which a deadline may be missed 11 times in every consecutive 295 jobs. It seems reasonable that systems that are built and certified (for example in the automotive domain) would not experience many deadline misses, and that using a short window size would lead to very conservative results.

To address these questions and empower researchers with a tool to apply their analysis techniques, this paper presents **WeaklyHard.jl**, a software library for weakly hard tasks that treats scalability as a first-class citizen. More precisely, the contributions of the paper are the following:

- We provide a theoretical contribution on the relation between weakly hard tasks that constrain the number of hits and the number of consecutive hits in a window (Section III). This relation allows us to relate all the types of constraints with one another, and provide some ordering among them.
- We leverage an automata-based representation to describe the behaviour of a task subject to a weakly-hard constraint [21], [30]. In constrast to other approaches, our description exploits a mapping between a single transition in the automaton and a deadline (Section IV). This enables uses such as automatic generation of monitors to check weakly-hard constraint satisfaction on the fly.
- We extend the automaton to describe a task subject to a finite set of weakly-hard constraints (Section III). In this way, we are able to address the analysis of systems that satisfy multiple constraints, possibly of different types, that do not dominate one another. As far as we know, this is the first paper that presents an analysis of a set of weakly-hard constraints.

We conduct an extensive performance evaluation campaign with a two-fold purpose (Section V). First, we analyse the scalability of our library compared to the state of the art whenever possible, i.e., for single constraints. Second, we look at sets of constraints and perform a sensitivity analysis, to determine which parameters affect the execution time of the automaton construction for a set of constraints.

II. Background and related work

In this work, we analyse a single real-time task. For the remainder of this paper, a real-time task τ is an entity composed of a sequence of jobs $(j_i)_{i \in \mathbb{N}^{\geq}}$, representing code that is executed repeatedly on a given hardware platform (not necessarily according to any temporal pattern or periodicity). A task is characterised by its relative deadline d, representing the time after which each job should be completed.

The index i counts the job number. For a given job j_i, we denote with a_i its release time (the time in which the job becomes active in the hardware platform), and with f_i its completion time (the time in which the job terminates its execution). We also use d_i to represent the absolute deadline of the i-th job, meaning that $d_i = a_i + d$.

In general, a job can either complete its execution before its deadline or overrun it, resulting respectively in a deadline *hit* or *miss* (collectively denoted by the job's *outcome*).

Definition 1 (Deadline Hit)**.** *The i-th job of a task τ is said to hit its deadline if $f_i \leq d_i$.*

Definition 2 (Deadline Miss)**.** *The i-th job of a task τ is said to miss its deadline if $f_i > d_i$.*

The weakly-hard task model [4], [5] provides guarantees on the sequence of outcomes of a real-time task via four constraints, each specifying how deadline misses and hits are interleaved for a window of $k \geq 1$ consecutive jobs.

Definition 3 (Weakly-Hard Task)**.** *A weakly-hard task τ is a task that satisfies (at least) one of the following constraints:*

(i) $\tau \vdash \binom{x}{k}$ ***(AnyHit)****: in any window of k consecutive jobs, the minimum number of hits is x;*
(ii) $\tau \vdash \left\langle {x \atop k} \right\rangle$ ***(RowHit)****: in any window of k consecutive jobs, the minimum number of consecutive hits is x;*
(iii) $\tau \vdash \overline{\binom{x}{k}}$ ***(AnyMiss)****: in any window of k consecutive jobs, the maximum number of misses is x; and*
(iv) $\tau \vdash \overline{\left\langle {x \atop k} \right\rangle}$ ***(RowMiss)****: in any window of k consecutive jobs, the maximum number of consecutive misses is x;*

for some values of $x \in \mathbb{N}^{\geq}$, $k \in \mathbb{N}^{>}$, where $x \leq k$. We use the $\vdash$ symbol to indicate that all the possible sequences of outcomes of τ satisfy the right hand side.

The types of constraints in Definition 3 have received different attention in the real-time systems literature. In particular, the **AnyMiss** constraint has been extensively studied, and is commonly addressed as the (m, K) weakly-hard task model [1], [12], [14], [15], [24], [28]. However, these constraints have been studied separately, while a task can simultaneously satisfy many, possibly of different types.

Exploiting different types of constraints – and possibly different parameters for the same type of constraint – leads to a better outcome for the analysis of the system. This follows from the space of possible sequences being pruned, thus allowing us to focus on proving that the real-time system behaves correctly in the relevant cases. In the following, we denote a set of L weakly-hard constraints with $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_L\}$.

To characterise the possible sequences of outcomes that satisfy a constraint, we borrow some elementary concepts from language theory, in particular the *binary alphabet* [16].

Definition 4 (Alphabet Σ of Job Outcomes)**.** *We define the alphabet of job outcomes $\Sigma = \{0, 1\}$, where 0 indicates a deadline miss and 1 represents a deadline hit.*

Using well-established notation, we denote the *character* $c_i \in \Sigma$ as the outcome of job j_i. A *word* w of length $|w| = N$ is a sequence of characters $w = (c_1, c_2, \dots, c_N)$ that specifies a sequence of consecutive job outcomes for a task. Without loss of generality, we assume that all words are preceded and followed only by hits. We denote the subword of a word w from index a to b with $w(a, b) = (c_a, c_{a+1}, \dots, c_b)$. Finally, Σ^N denotes the set of all possible words of length N.

With a slight abuse of notation, we use $w \vdash \lambda$ to indicate that the word w satisfies the constraint λ. Obtaining the set of words satisfying λ follows directly from the definitions of the alphabet and the constraint itself [4], [5].

Definition 5 (Satisfaction Set $\mathcal{S}_N(\lambda)$)**.** *The set of all length N words w, satisfying the weakly-hard constraint λ, is denoted by $\mathcal{S}_N(\lambda)$. Formally, $\mathcal{S}_N(\lambda) = \left\{w \in \Sigma^N \mid w \vdash \lambda\right\}, N \geq 1$.*

Trivially, all words in $\mathcal{S}_M(\lambda)$ are subwords of words existing in $\mathcal{S}_N(\lambda)$, if $M \leq N$. To simplify notation we define the set containing all words of infinite length as $\mathcal{S}(\lambda) \equiv \mathcal{S}_\infty(\lambda)$.

Using satisfaction sets, it is possible to formally define a partial ordering between two constraints λ_i and λ_j. We denote the logical conjunction with $\wedge$ and the logical disjunction with $\vee$. The following notions of constraint *domination* and *equivalence* [4], [5] are used extensively throughout the remainder of the paper (jointly denoted *constraint dominance*).

Definition 6 (Constraint Domination)**.** *Given two arbitrary weakly-hard constraints λ_i and λ_j, λ_i dominates λ_j (denoted $\lambda_i \prec \lambda_j$) if all words satisfying λ_i also satisfy λ_j, i.e., $\mathcal{S}(\lambda_i) \subset \mathcal{S}(\lambda_j)$. Correspondingly, $\lambda_i \preceq \lambda_j \Leftrightarrow \mathcal{S}(\lambda_i) \subseteq \mathcal{S}(\lambda_j)$.*

Definition 7 (Constraint Equivalence)**.** *Given two arbitrary weakly-hard constraints λ_i and λ_j, λ_i is equivalent to λ_j if they respectively dominate each other. Formally, $\lambda_i \equiv \lambda_j \Leftrightarrow \lambda_i \preceq \lambda_j \wedge \lambda_j \preceq \lambda_i$. Two constraints are equivalent if they share the same satisfaction set, i.e., $\lambda_i \equiv \lambda_j \Leftrightarrow \mathcal{S}(\lambda_i) = \mathcal{S}(\lambda_j)$.*

The notion of constraint dominance has attracted attention from different areas, and is still occasionally researched [29], [34]. To provide dominance results, we first define the *weakest* and *hardest* constraints [4], [5].

Definition 8 (Weakest Constraint $\underline{\lambda}$)**.** *The weakest constraint $\underline{\lambda}$ is defined as the constraint satisfied by* any *word. Formally, $\mathcal{S}_N(\underline{\lambda}) = \Sigma^N$, $\forall N \in \mathbb{N}^>$.*

Definition 9 (Hardest Constraint $\overline{\lambda}$)**.** *The hardest constraint $\overline{\lambda}$ is defined as the constraint satisfied* solely *by the word containing all deadline hits. Formally, $\mathcal{S}_N(\overline{\lambda}) = \left\{1^N\right\}, \forall N \in \mathbb{N}^>$.*

Using these definitions, we now review known constraint dominance relations. We refer the reader to [4] or any referenced paper for the corresponding proofs.

Lemma 1 (Known Equivalence Relations)**.** *The following equivalence relations hold:*

(i) $\binom{x}{k} \equiv \overline{\binom{k-x}{k}}$, *an **`AnyHit`** constraint with x deadline hits in a window of k jobs is equivalent to an **`AnyMiss`** constraint with $k-x$ hits in a window of k jobs,*

(ii) $\overline{\left\langle{x \atop k}\right\rangle} \equiv \overline{\langle x \rangle}$, $\forall k \geq 1$, *a **`RowMiss`** constraint is independent of the window size, i.e., it is equivalent to the same constraint with any k value,*

(iii) $\overline{\langle x \rangle} \equiv \overline{\binom{x}{x+1}}$, *a **`RowMiss`** constraint with x deadline misses is equivalent to an **`AnyMiss`** with x possible misses in a window of $x+1$ jobs [22],*

(iv) $\left\langle{1 \atop k}\right\rangle \equiv \binom{1}{k}$, *(trivially) a **`RowHit`** constraint is equivalent to an **`AnyHit`** when looking at the same window length and a single deadline,*

(v) $\left\langle{x \atop k}\right\rangle \equiv \overline{\lambda} \Leftrightarrow x > {}^{k}\!/_{2}$, *a **`RowHit`** constraint is equivalent to the hardest constraint when $x > {}^{k}\!/_{2}$.*

Using these equivalence relations, we can always translate **`AnyMiss`** and **`RowMiss`** constraints into a corresponding **`AnyHit`** constraint. However, there is no clear equivalence between **`AnyHit`** and **`RowHit`** constraints (beside the trivial case of a single deadline and the same window length). Finding such a relation is important because it would allow us to treat sets of different types of constraints reducing the analysis to a single type and therefore improving efficiency. This motivates our formal analysis of the relation between hit-related constraints, presented in Section III.

Denoting with $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ respectively the floor and ceiling operators, we can then define some domination relations.

Lemma 2 (Known Domination Relations)**.** *The following domination relations hold:*

(i) $\binom{x_1}{k_1} \preceq \binom{x_2}{k_2} \Leftrightarrow x_2 \leq \max\{a, b\}$, *where $a = \lfloor\frac{k_2}{k_1}\rfloor x_1$ and $b = k_2 - \lceil\frac{k_2}{k_1}\rceil(k_1 - x_1)$; the **`AnyHit`** constraint with parameters x_1 and k_1 dominates all **`AnyHit`** constraints with parameters x_2 and k_2 if and only if $x_2 \leq \max\{a, b\}$ with a and b defined as above.*

(ii) *For any two constraints* $\left\langle{x_1 \atop k_1}\right\rangle, \left\langle{x_2 \atop k_2}\right\rangle \not\equiv \overline{\lambda}$, $\left\langle{x_1 \atop k_1}\right\rangle \preceq \left\langle{x_2 \atop k_2}\right\rangle \Leftrightarrow \left(k_2 < k_1 \wedge k_2 \leq x_1 - \lceil\frac{k_1 - k_2}{2}\rceil\right) \vee (k_2 \geq k_1 \wedge x_2 \leq x_1)$; *this specifies the domination between two **`RowHit`** constraints depending on their constraint parameters.*

(iii) $\left\langle{x_1 \atop k}\right\rangle \preceq \binom{x_2}{k} \Rightarrow \{x_2 \leq 4x_1 - k - 2,\ x_2 \leq x_1,\ x_2 \geq 0\}$; *for a fixed and equal window k, if a **`RowHit`** constraint with consecutive deadlines hits x_1 dominates an **`AnyHit`** constraint with x_2 deadlines hits, then the indicated relation between the constraint parameters hold.*

(iv) $\overline{\langle x_1 \rangle} \preceq \overline{\langle x_2 \rangle} \Leftrightarrow x_1 \leq x_2$; *a **`RowMiss`** constraint with a lower number of deadline misses dominates a **`RowMiss`** with a higher number of deadline misses.*

(v) $\overline{\binom{x+p}{k+p}} \preceq \overline{\binom{x}{k}}$ *if $p > 0$; **`AnyMiss`** constraints can be dominated by other **`AnyMiss`** constraints when particular relations hold for values of their parameters [29].*

The ability to translate constraints into **AnyHit** equivalents makes Lemma 2(i) very powerful to compare different weakly hard constraints. Finally, Lemma 2(iii) is the only known result that relates the **RowHit** constraints with the other types. However, its applicability is limited to the case in which the two constraints share the same window size. From the presentation of the existing constraint dominance relations, we gather that there is an important piece missing to achieve a comprehensive weakly-hard analysis.

III. **AnyHit**, **RowHit**, AND CONSTRAINT SETS

This section contains the theoretical contribution of the paper. In III-A, we present some novel results on the relation between the **RowHit** and **AnyHit** constraints. The results introduce the final theoretical pieces allowing us to relate all the weakly-hard constraint types to the **AnyHit** constraint, and thus to pave the way towards an efficient analysis implementation. In III-B, we extend the theoretical results to handle sets of constraints, possibly containing constraints of different types.

*A. Relating **RowHit** and **AnyHit** constraints*

Our first theoretical contribution is the proof of a condition regarding the domination of a **RowHit** constraint over a **AnyHit** constraint, precisely

$$\left\langle {x_1 \atop k_1} \right\rangle \preceq \binom{x_2}{k_2} \Leftrightarrow x_2 \leq x_1 \lfloor k_2/p \rfloor + \max\{0, x_1 - p + (k_2 \bmod p)\}$$

with $p = k_1 - x_1 + 1$. The proof is based on restricting the **AnyHit** constraint's minimum number of hits in order to ensure that its satisfaction set includes the one of the **RowHit** constraint.

Theorem 1 (**RowHit**–**AnyHit** Domination)**.** *Let $\mathcal{S}$ be the satisfaction set of the **RowHit** constraint $\lambda_1 = \left\langle {x_1 \atop k_1} \right\rangle$, and $k_2 \geq x_2$ be non-negative integers. Then the following are equivalent:*

*(i) Every sequence in $\mathcal{S}$ satisfies the **AnyHit** constraint $\binom{x_2}{k_2}$;*
(ii) $x_2 \leq x_1 \lfloor k_2/p \rfloor + \max\{0, x_1 - p + (k_2 \bmod p)\}$, where $p = k_1 - x_1 + 1$.

Proof. ¬(ii) ⇒ ¬(i): Consider the binary sequence that alternates between x_1 consecutive 1's and $p - x_1$ consecutive 0's, where p is as in *(ii)*:

$$\bar{s} = \ldots \underbrace{1\ldots1}_{x_1}\overbrace{0\ldots0}^{p-x_1}\underbrace{1\ldots1}_{x_1}\overbrace{0\ldots0}^{p-x_1}\ldots \tag{1}$$

First observe that $\bar{s} \in \mathcal{S}$. Since $k_2 = \lfloor k_2/p \rfloor + (k_2 \bmod p)$, $\bar{s}$ contains a sub-string of length k_2 with

$$x_1 \lfloor k_2/p \rfloor + \max\{0, x_1 - p + (k_2 \bmod p)\}$$

1's. Therefore, if the inequality in *(ii)* does not hold, then $\bar{s}$ does not satisfy the **AnyHit** constraint $\lambda_2 = \binom{x_2}{k_2}$, since this sub-string would contain fewer than x_2 1's.

(ii) ⇒ (i): Let s be any sequence in $\mathcal{S}$. Now let s' be equal to s, except that every maximal sub-string of 1's with fewer than x_1 elements has been replaced with a sub-string of zeros:

$$s'_i = \begin{cases} 1 & \text{if } s_i \text{ is part of a sub-string of at least } x_1 \text{ 1's}, \\ 0 & \text{otherwise.} \end{cases}$$

First observe that $s' \in \mathcal{S}$. This is because maximal sub-strings of 1's with fewer than x_1 elements do not contribute to the satisfaction of a **RowHit** constraint (from the perspective of this constraint, such sub-strings may as well be zeros). Also note that if s' satisfies an **AnyHit** constraint, so does s. This is because s can be obtained from s' by flipping some 0's to 1's, which cannot lead to a violation of an **AnyHit** constraint. Therefore, it is sufficient to show that if *(ii)* holds, any such s' satisfies the **AnyHit** constraint in *(i)*. By construction, s' alternates between sub-strings of 1's with at least x_1 elements, and sub-strings of zeros of at most $p - x_1$ elements

$$s' = \ldots \underbrace{1\ldots1}_{\geq x_1}\ \overbrace{0\ldots0}^{\leq p-x_1}\ \underbrace{1\ldots1}_{\geq x_1}\ \overbrace{0\ldots0}^{\leq p-x_1}\ \ldots$$

It then follows that every sub-string of length k_2 in s' has at least as many 1's as every sub-string of length k_2 in the sequence $\bar{s}$ from (1). Since $\bar{s}$ satisfies the **AnyHit** constraint, so does s', and therefore so does every $s \in \mathcal{S}$ as required. □

The second theoretical contribution of the paper is the proof of a condition regarding the domination of an **AnyHit** constraint over a **RowHit** constraint, specifically

$$\binom{x_1}{k_1} \preceq \left\langle {x_2 \atop k_2} \right\rangle \Leftrightarrow x_2 \leq \min\{\lfloor k_2/(z_1+1) \rfloor, \lceil x_1/z_1 \rceil\}$$

where $z_1 = k_1 - x_1$.

Theorem 2 (**AnyHit**–**RowHit** Domination)**.** *Let $\mathcal{S}$ be the satisfaction set of the **AnyHit** constraint $\binom{x_1}{k_1}$, and $k_2 \geq x_2$ be non-negative integers. Then the following are equivalent:*

*(i) Every sequence in $\mathcal{S}$ satisfies the **RowHit** constraint $\left\langle {x_2 \atop k_2} \right\rangle$;*
(ii) $x_2 \leq \min\{\lfloor k_2/(z_1+1) \rfloor, \lceil x_1/z_1 \rceil\}$, where $z_1 = k_1 - x_1$.

Proof. ¬(ii) ⇒ ¬(i): We split the proof into three cases.

Case 1: $0 < k_2 \leq z_1$. Let $\bar{s} = \ldots s_d s_d s_d \ldots$ (i.e. the sequence constructed by repeating the sub-string s_d), where

$$s_d = \underbrace{1\ldots1}_{x_1}\overbrace{0\ldots0}^{z_1}.$$

Observe that $\bar{s} \in \mathcal{S}$. Since $\lfloor k_2/(z_1+1) \rfloor = 0$, ¬*(ii)* implies that $x_2 > 0$. This implies ¬*(i)* because $\bar{s}$ contains at least k_2 consecutive 0's, and therefore cannot satisfy the **RowHit** constraint $\left\langle {x_2 \atop k_2} \right\rangle$.

Case 2: $k_2 > z_1 \wedge \lceil x_1/z_1 \rceil \geq \lfloor k_2/(z_1+1) \rfloor$. Let s_d be a sequence of length k_2 consisting of $k_2 - z_1$ 1's and z_1 0's, with the 1's arranged into $z_1 + 1$ sub-strings

$$s_d = \underbrace{1\ldots1}_{l_1}0\underbrace{1\ldots1}_{l_2}0\ldots0\underbrace{1\ldots1}_{l_{z_1+1}},$$

where the lengths of the sub-strings l_k satisfy

$$l_k \in \{ \left\lfloor \frac{k_2 - z_1}{z_1 + 1} \right\rfloor, \left\lceil \frac{k_2 - z_1}{z_1 + 1} \right\rceil \}.$$

Let $\bar{s} = \ldots 111 s_d 111 \ldots$ (i.e. a sequence of all 1's except for a single sub-string s_d). Since this sequence contains only z_1 0's, $\bar{s} \in \mathcal{S}$. The conclusion now follows since

$$\left\lceil \frac{k_2 - z_1}{z_1 + 1} \right\rceil = \left\lfloor \frac{k_2 - z_1 - 1}{z_1 + 1} \right\rfloor + 1 = \left\lfloor \frac{k_2}{z_1 + 1} \right\rfloor,$$

and so if $x_2 > \lfloor k_2 / (z_1 + 1) \rfloor$, then this $\bar{s}$ does not satisfy the **RowHit** constraint $\left\langle {x_2 \atop k_2} \right\rangle$.

Case 3: $k_2 > z_1 \wedge \lceil x_1/z_1 \rceil < \lfloor k_2 / (z_1 + 1) \rfloor$. Let s_d be a sequence of length k_1 consisting of x_1 1's and z_1 0's, with the 1's arranged into z_1 sub-strings

$$s_d = \underbrace{1 \ldots 1}_{l_1} 0 \underbrace{1 \ldots 1}_{l_2} 0 \ldots 0 \underbrace{1 \ldots 1}_{l_{z_1}} 0,$$

where the lengths of the sub-strings l_k satisfy $l_k \in \{\lfloor x_1/z_1 \rfloor, \lceil x_1/z_1 \rceil\}$. Let $\bar{s} = \ldots s_d s_d s_d \ldots$ (i.e. the sequence constructed by repeating the sub-string s_d). Observe that every sub-string of length k_1 in $\bar{s}$ contains exactly x_1 1's, and therefore $\bar{s} \in \mathcal{S}$. Observe also that $\bar{s}$ contains no sub-strings of more than $\lceil x_1/z_1 \rceil$ consecutive 1's, and therefore if $x_2 > \lceil x_1/z_1 \rceil$, $\bar{s}$ does not satisfy the **RowHit** constraint $\left\langle {x_2 \atop k_2} \right\rangle$.

¬(i) ⇒ ¬(ii): Under the hypothesis of *¬(i)*, there exists a sequence $s \in \mathcal{S}$ such that s does not satisfy the **RowHit** constraint $\left\langle {x_2 \atop k_2} \right\rangle$. Let s' be the sequence obtained from s by removing all 0's from the start of s, and then replacing all sub-strings of 0's with length greater than one with a single 0 (for example, if $s = 011001010001\ldots$, then $s' = 11010101\ldots$. Clearly $s' \in \mathcal{S}$ since this process only removes 0's, and s' also does not satisfy the **RowHit** constraint. Consider now the sub-string s_d formed from the first k_2 elements of s'.[1] This sub-string will take the form

$$s_d = \begin{cases} \underbrace{1 \ldots 1}_{l_1} 0 \underbrace{1 \ldots 1}_{l_2} 0 \ldots 0 \underbrace{1 \ldots 1}_{l_n}, & \text{or} \\ \underbrace{1 \ldots 1}_{l_1} 0 \underbrace{1 \ldots 1}_{l_2} 0 \ldots 0 \underbrace{1 \ldots 1}_{l_n} 0, \end{cases}$$

depending on whether the final element is 0 or 1. Note that the lengths of the sub-strings of 1's satisfy $0 \le l_k < x_2$. We will now show that the existence of such a sub-string implies *¬(ii)* by considering two cases.

Case 1: $k_2 \le k_1 + l_1$. In this case the sub-string s_d contains at most z_1 0's, and so $n \le z_1 + 1$. The pigeonhole principle then demonstrates that there must be an integer $1 \le k \le n$ such that

$$l_k \ge \left\lceil \frac{k_2 - z_1}{n} \right\rceil.$$

[1]Strictly speaking if s is too short, then the sequence s' resulting from this process might have length less than $\min\{k_1, k_2\}$ which would mean that the statement $s' \in \mathcal{S}$ is ill defined. In this case 0's should only be removed until s' has length $\min\{k_1, k_2\}$. This will still result in a sequence that satisfies the **AnyHit** constraint but violates the **RowHit** constraint. All the given arguments remain valid for such an s', since they only depend on inequalities based on the number of 0's in particular sub-strings of length k_1 as guaranteed by the **AnyHit** constraint (note in *Case 1* it is perfectly valid for $l_1 = 0$).

To see this, note that s_d has at least $k_2 - z_1$ 1's, and these must be allocated into n pigeonholes corresponding to the n sub-strings of 1's. This implies that

$$x_2 > l_k \ge \left\lceil \frac{k_2 - z_1}{n} \right\rceil \ge \left\lceil \frac{k_2 - z_1}{z_1 + 1} \right\rceil = \left\lfloor \frac{k_2}{z_1 + 1} \right\rfloor$$

and $\lfloor k_2/(z_1+1) \rfloor \ge \min\{\lfloor k_2/(z_1+1) \rfloor, \lceil x_1/z_1 \rceil\}$ as required.

Case 2: $k_2 > k_1 + l_1$. Let s'_d denote the sub-string obtained by removing the first l_1 elements of s_d, and also removing elements from the end of s_d, until s'_d has length k_1. This sub-string takes the form

$$s'_d = \begin{cases} 0 \underbrace{1 \ldots 1}_{l_2} 0 \underbrace{1 \ldots 1}_{l_3} 0 \ldots 0 \underbrace{1 \ldots 1}_{l_{m+1}}, & \text{or} \\ 0 \underbrace{1 \ldots 1}_{l_2} 0 \underbrace{1 \ldots 1}_{l_3} 0 \ldots 0 \underbrace{1 \ldots 1}_{l_{m+1}} 0, \end{cases}$$

depending on whether the final element is 0 or 1. Since s'_d satisfies the **AnyHit** constraint $\binom{x_1}{k_1}$, it contains at most z_1 zeros, and so $m \le z_1$. Therefore, in this case the pigeonhole principle implies that at least one of the lengths l_k must satisfy

$$l_k \ge \left\lceil \frac{x_1}{m} \right\rceil \ge \left\lceil \frac{x_1}{z_1} \right\rceil.$$

This implies $x_2 > l_k \ge \lceil x_1/z_1 \rceil \ge \min\{\lfloor k_2/(z_1+1) \rfloor, \lceil x_1/z_1 \rceil\}$ as required. □

The two theorems above complete the relation graph between the different types of weakly-hard constraints. Now that we have a complete picture, we can start investigating sets Λ of L constraints, $\Lambda = \{\lambda_1, \ldots, \lambda_L\}$.

B. *Handling sets of weakly-hard constraints* Λ

We extend the theory to the case in which τ is subject to an arbitrary set of constraints of the form presented in Definition 3. First, we extend the satisfaction from Definition 5 and obtain

$$\mathcal{S}_N(\Lambda) = \bigcap_{\lambda \in \Lambda} \mathcal{S}_N(\lambda) \qquad (2)$$

where $\bigcap$ is the generalised intersection. We use $\tau \vdash \Lambda$ to denote that τ satisfies all the constraints in the set Λ. This implies that each word $w \in \mathcal{S}_N(\Lambda)$ must belong to the satisfaction set of all the constraints in Λ. Trivially, Equation (2) allows us to extended Definitions 5 and 6 to define constraint dominance for sets of constraints.

Constraint dominance significantly reduces the problem complexity when working with sets of weakly-hard constraints, Λ. If the constraint set supports different types of weakly-hard constraints, it can be beneficial to find an equivalent set of constraints with minimal cardinality.

To minimise the number of constraints in the problem formulation, the constraint dominance is utilised in order to find the minimal cardinality, equivalent subset. Utilising the comprehensive picture the theorems provide, we propose the notion of a *dominant set*, thus simplifying the analysis of weakly-hard systems subject to multiple constraints.

Definition 10 (Dominant Set)**.** *The dominant set Λ^* of a set of weakly-hard constraints Λ is defined as the smallest cardinality subset of Λ representing an equivalent set of constraints. Formally, $\Lambda^* \subseteq \Lambda$ where*

(i) $\lambda_i, \lambda_j \in \Lambda^* \;\Rightarrow\; \lambda_i \not\equiv \lambda_j,\ \forall i \neq j,$
(ii) $\lambda_i, \lambda_j \in \Lambda^* \;\Rightarrow\; \lambda_i \not\preceq \lambda_j,\ \forall i \neq j,$
(iii) $\lambda_i \in \Lambda \setminus \Lambda^* \;\Rightarrow\; \exists \lambda_j \in \Lambda^*\ s.t.\ \lambda_j \preceq \lambda_i.$

From Definition 6, a weakly-hard constraint λ_i dominates λ_j if and only if $\mathcal{S}(\lambda_i) \subseteq \mathcal{S}(\lambda_j)$. Thus, excluding all the dominated constraints from Λ does not change the resulting satisfaction set. The equivalence between the constraint set and its dominant set is trivial considering the respective satisfaction sets:

$$\mathcal{S}(\Lambda^*) = \bigcap_{\lambda \in \Lambda^*} \mathcal{S}(\lambda) = \bigcap_{\lambda \in \Lambda} \mathcal{S}(\lambda) = \mathcal{S}(\Lambda).$$

In the following section, we present our tool, **`WeaklyHard.jl`**, and use the theorems presented in this section and the dominance between constraints to simplify the analysis of sets of weakly-hard constraints.

IV. `WeaklyHard.jl`

In this section we introduce **`WeaklyHard.jl`**[2], a scalable tool for analysing (sets of) weakly-hard constraints of different types. The tool facilitates the analysis of weakly-hard tasks providing functions to:

(i) compare two arbitrary weakly-hard constraints or two sets of weakly-hard constraints, obtaining answers about their dominance,
(ii) translate a weakly-hard constraint or a set of weakly-hard constraints into a corresponding automaton, that represents all the sequences that belong to the satisfaction set of the set of constraints,
(iii) produce *all* sequences of arbitrary length that satisfy a set of weakly-hard constraints, i.e., the satisfaction set.

We distribute **`WeaklyHard.jl`** as an open-source package, written in the Julia programming language [6]. Julia is a scripting language with Just-In-Time compilation. The language design is centered upon two core concepts: type-stability and function specialisation through multiple-dispatch. The type-stable compilation provides an implementation that is close to the hardware, resulting in efficient code execution. Multiple-dispatching allows us to write a user-friendly code library. Additionally, Julia's built in package manager simplifies the distribution of non-proprietary packages.

A task subject to any weakly-hard constraint (from Definition 3) can be represented using an automaton. Automata have been used in the analysis of networked systems [18], [31], schedulability [10], [11], [36], and control systems [20], [21], [24], [30]. In this paper, we decided to constrain the automaton structure, thinking about the possible use of the automaton, e.g., generating a monitor to check whether a constraint is satisfied. In our representation, vertices encode the task's state, i.e., the relevant suffix of the sequence of job outcomes. Similarly, edges are associated with a *feasible* outcome (hit or miss) and encode the transitions from one state to another. Feasibility here refers to the fact that deadline misses are not allowed if the constraint would not permit them. The outcome sequences acquired from *all* random walks in the automaton correspond to the satisfaction set of the weakly-hard constraint represented by the automaton.

Due to their combinatorial nature, weakly-hard systems are inherently complicated to analyse. Their complexity becomes apparent in the size of the automaton, and evidently grows when the window length of the constraint increases. In the following, we present a scalable approach for generating automata representations of weakly-hard constraints.

A. Weakly-hard constraints as automata

Suppose that $\tau \vdash \lambda$. We use $\mathcal{G}_\lambda = (V_\lambda, E_\lambda)$ to indicate the directed labeled graph $\mathcal{G}_\lambda$ corresponding to the automaton representation of τ. Here, V_λ represents the set of *vertices* in the graph and E_λ represents the directed *edges* between vertices (also denoted *transitions*). Each vertex $v_i \in V_\lambda$ represents a word $w_i \in \mathcal{S}(\lambda)$. With a slight notational abuse, vertices v_i will occasionally (when evident from context) be treated as the word they represent, w_i. The transition $e_{i,j} \in E_\lambda$ corresponds to a tuple $e_{i,j} = (v_i, v_j, c_{i,j})$, where the vertex pair $v_i, v_j \in V_\lambda$ denotes the tail and head of the transition, and the character $c_{i,j} \in \Sigma$ corresponds to the transition's label. A transition $e_{i,j}$ is feasible if and only if the concatenation of the character $c_{i,j}$ to the word w_i satisfies λ. Formally:

$$e_{i,j} \in E_\lambda \Leftrightarrow (w_i(2, |w_i|), c_{i,j}) = w_j \vdash \lambda.$$

Finally, for two vertices $v_i, v_j \in V_\lambda$ we say that v_j is a direct successor of v_i if there exists a transition $e_{i,j} \in E_\lambda$. Without loss of generality, we will assume that each vertex $v_i \in V_\lambda$ can have at most two direct successors with distinct transition outcomes, i.e., one successor v_{j_1} through $e_{i,j_1} = (v_i, v_{j_1}, 1)$ and (if permissible) one successor v_{j_0} through $e_{i,j_0} = (v_i, v_{j_0}, 0)$.

B. Automaton construction

The naïve approach of constructing the automaton $\mathcal{G}_\lambda$ is both time consuming and memory intensive (including $|\mathcal{S}_k(\lambda)|$ vertices, where k is the window length of λ). In order to improve performance and scalability, we include the following optimisations:

(i) representing words as bit strings,
(ii) minimising the automata size by combining equivalent vertices during the automata generation, and
(iii) representing large sets of constraints with their dominant subset.

Support for bit string operations (like shifting) is essential for efficient sequence management. Logical and bitwise operations are directly supported by all processors, thus they are highly optimised and require a minimal amount of instruction cycles. We use the following notation: $\&$ is the *bitwise and*, $|$ is the *bitwise or*, and $\ll$ is the *logical left-shift*.

[2]**`WeaklyHard.jl`** is available from the track chair upon request, and will be released as open source together with this publication.

Algorithm 1 Generation of the minimal automaton representation $\mathcal{G}_\lambda$ corresponding to a weakly-hard constraint λ.

```
1: procedure BUILDAUTOMATON($\lambda$)
2:     $V_\lambda \leftarrow \{v_1 = (1 \ll n) - 1\}$
3:     $E_\lambda \leftarrow \emptyset$, $Q = \{v_1\}$
4:     while $Q \neq \emptyset$ do
5:         $v_i \leftarrow pop\,(Q)$
6:         $v_{j_0} \leftarrow compact\,(\lambda,\ (v_i \ll 1) \mid 0)$
7:         $v_{j_1} \leftarrow compact\,(\lambda,\ (v_i \ll 1) \mid 1)$
8:         if $v_{j_0} \vdash \lambda$ then
9:             if $v_{j_0} \notin V_\lambda$ then
10:                $V_\lambda \leftarrow V_\lambda \cup \{v_{j_0}\}$
11:                $Q \leftarrow Q \cup \{v_{j_0}\}$
12:            $E_\lambda \leftarrow E_\lambda \cup \{e_{i,j_0} = (v_i, v_{j_0}, 0)\}$
13:        if $v_{j_0} \notin V_\lambda$ then
14:            $V_\lambda \leftarrow V_\lambda \cup \{v_{j_1}\}$
15:            $Q \leftarrow Q \cup \{v_{j_1}\}$
16:        $E_\lambda \leftarrow E_\lambda \cup \{e_{i,j_1} = (v_i, v_{j_1}, 1)\}$
       return $\mathcal{G}_\lambda = (V_\lambda, E_\lambda)$
```

Each word $w \in \mathcal{S}(\lambda)$ is a sequence of outcomes and can therefore be interpreted as a string of bits – recall that an outcome is a character in $\Sigma = \{0, 1\}$. The rightmost character in w is the outcome of the last job, e.g., $w = 001$ implies that the last deadline was hit, but the two previous ones were missed. Assuming that the task τ experienced the outcomes w and the next outcome is $c \in \Sigma$, then the new sequence of outcomes is $w' = (w \ll 1) \mid c$.

The size of the naïve automaton can be reduced substantially by combining vertices that would otherwise result in language-equivalent states [16]. Two vertices $v_{i_1}, v_{i_2} \in V_\lambda$ are considered equivalent if they share the same direct successors with the same transition outcomes. As an example, consider the **AnyHit** constraint $\lambda = \binom{1}{2}$. Trivially there are only three feasible vertices in the naïve automaton, since there are $2^k = 4$ words in Σ^k and $w = 00$ is infeasible. The words $w_1 = 11$ and $w_2 = 01$ are equivalent since they share the same direct successors with the same transition outcomes, i.e., $(w_1 \ll 1) \,|\, 0 = (w_2 \ll 1) \,|\, 0$ and $(w_1 \ll 1) \,|\, 1 = (w_2 \ll 1) \,|\, 1$, considering the window $k = 2$. Intuitively, the fact that it is possible to combine vertices comes from the realisation that a task's history, prior to the last k job outcomes, is irrelevant. Combining the equivalent vertices results in a new vertex representing the word $w = w_1 \,\&\, w_2$.

Finally, for sets of weakly-hard constraints Λ we construct the graph $\mathcal{G}_{\Lambda^*}$ for the dominant set $\Lambda^* \subseteq \Lambda$. Since $\mathcal{S}(\Lambda^*) = \mathcal{S}(\Lambda)$, it also follows that $\mathcal{G}_{\Lambda^*} \equiv \mathcal{G}_\Lambda$.

We generate the minimal automaton $\mathcal{G}_\lambda$ as presented in Algorithm 1. The automaton is initialised with a single vertex corresponding to the word $w_1 = 1^n$, $v_1 = (1 \ll n) - 1$. Here, n is the smallest number of hits required in a window to meet the constraint λ, e.g., $n = 1$ for $\lambda = \overline{\langle 3 \rangle}$ or $n = 2$ for $\left\langle {2 \atop 5} \right\rangle$. As long as there exists uninitialised vertices v_i, its successors v_{j_0} and v_{j_1} are created and passed through a function in order to *compact* them. This step reduces the new word to the minimal, equivalent word that would still satisfy λ. In particular, if either $(v_i \ll 1) \,|\, 0$ or $(v_i \ll 1) \,|\, 1$ return an existing vertex v_{i_0} or v_{i_1}, then v_{j_0} and v_{j_1} are reduced to the corresponding existing one. If the resulting words would satisfy λ, they are properly added to the automaton. Note that it is only required to verify that the successor following a deadline miss satisfy the constraint.

C. Scalable automata generation

Intuitively, the time required for generating an automaton is directly correlated to its size, i.e., more vertices lead to a larger exploration time and hence to a larger automaton-construction time. Additionally, the automata-based representation can be used in embedded devices, e.g., to monitor the satisfaction of a constraint. Thus, space and memory requirements create a clear need for the automaton to be minimal.

We provide a brief discussion on the minimum number of vertices needed to express the automaton corresponding to the weakly hard constraints presented in Definition 3. The structure of the minimal automaton depends on the type of constraint. For example, to describe an **AnyHit** constraint $\binom{x_{ah}}{k_{ah}}$ we need to keep track of the number and the position of the deadline hits we encountered in the past k_{ah} outcomes, giving us a number of vertices that corresponds to the binomial coefficient k_{ah} *choose* x_{ah}. The **AnyMiss** constraint can be reduced to the **AnyHit** constraint and hence we easily obtain the number of its vertices. For the **RowMiss** constraint, the number of vertices is also obvious, as we need to count the number of consecutive deadlines that have been missed, and return to the initial state as soon as the following outcome is a hit. Denoting with $\mathrm{s}(\lambda)$ the function that counts the number of vertices of the minimal automaton corresponding to the constraint λ, we obtain:

$$\textbf{AnyHit}: \quad \lambda_{ah} = \binom{x_{ah}}{k_{ah}} \Rightarrow \mathrm{s}(\lambda_{ah}) = \frac{k_{ah}!}{x_{ah}!\,(k_{ah} - x_{ah})!}$$

$$\textbf{AnyMiss}: \quad \lambda_{am} = \overline{\binom{x_{am}}{k_{am}}} \Rightarrow \mathrm{s}(\lambda_{am}) = \frac{k_{am}!}{x_{am}!\,(k_{am} - x_{am})!}$$

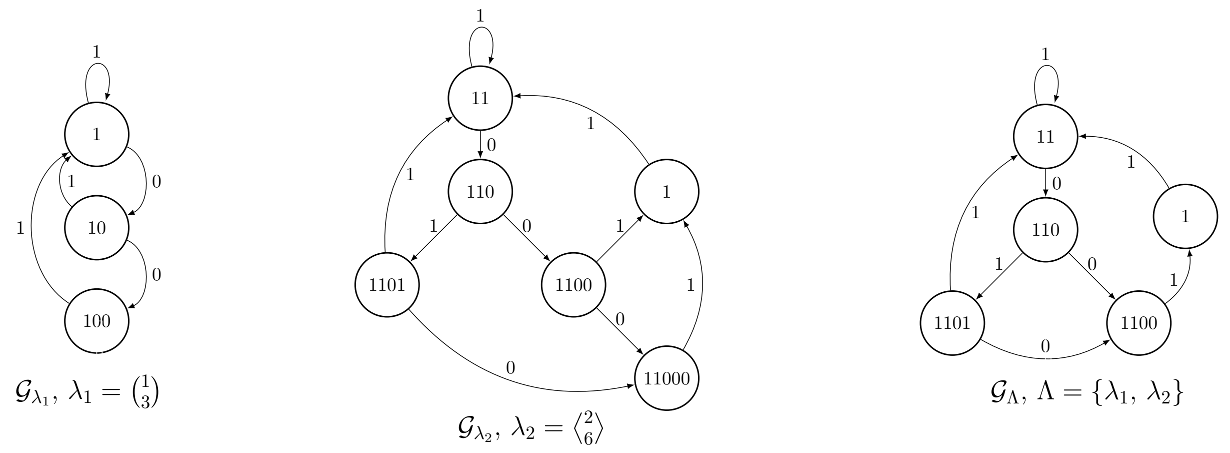
$$\textbf{RowMiss}: \quad \lambda_{rm} = \overline{\left\langle {x_{rm} \atop k_{rm}} \right\rangle} \Rightarrow \mathrm{s}(\lambda_{rm}) = x_{rm} + 1$$

e.g., the minimal automaton for the **AnyMiss** constraint $\overline{\binom{5}{20}}$ includes 15 504 vertices.

The **RowHit** constraint, $\left\langle {x_{rh} \atop k_{rh}} \right\rangle$ is more interesting. When $k_{rh} < 2\,x_{rh}$, the constraint reduces to the hardest constraint $\overline{\lambda}$, hence the automaton has a single vertex. If $k_{rh} = 2\,x_{rh}$, it is possible to have a single deadline miss, that can only appear before a sequence of x_{rh} has been recorded, hence the corresponding automaton has $x_{rh} + 1$ vertices. If $k_{rh} = 2\,x_{rh} + 1$, the number of vertices of the automaton are $x_{rh} + 2$ and subsequent values can be found using recursion. Specifically,

$$\textbf{RowHit}: \lambda_{rh} = \left\langle {x_{rh} \atop k_{rh}} \right\rangle \Rightarrow \mathrm{s}(\lambda_{rh}) = \begin{cases} 1 & k_{rh} < 2x_{rh} \\ x_{rh} + 1 & k_{rh} = 2x_{rh} \\ x_{rh} + 2 & k_{rh} = 2x_{rh} + 1 \\ 2\,\mathrm{s}\left(\left\langle {x_{rh} \atop k_{rh}-1} \right\rangle\right) - \mathrm{s}\left(\left\langle {x_{rh} \atop k_{rh}-2} \right\rangle\right) + 1 & 2x_{rh} + 1 < k_{rh} < 3x_{rh} \\ \mathrm{s}\left(\left\langle {x_{rh} \atop k_{rh}-1} \right\rangle\right) + x_{rh} & k_{rh} \geq 3x_{rh}. \end{cases}$$

In contrast to the **AnyHit** or **AnyMiss** constraints, the size of the minimal automaton corresponding to the **RowHit** constraint



$\mathcal{G}_{\lambda_1},\ \lambda_1 = \binom{1}{3}$
$\mathcal{G}_{\lambda_2},\ \lambda_2 = \left\langle{2 \atop 6}\right\rangle$
$\mathcal{G}_{\Lambda},\ \Lambda = \{\lambda_1,\ \lambda_2\}$
1
10
100
11
110
1101
1100
11000

Fig. 1. Minimal automata $\mathcal{G}_{\lambda_1}$, $\mathcal{G}_{\lambda_2}$, and $\mathcal{G}_{\Lambda}$ representing respectively λ_1, λ_2, and $\Lambda = \{\lambda_1, \lambda_2\}$ from Example 1.

is linear in the window length k_{rh} in stationarity, i.e., when $k_{rh} \geq 3x_{rh}$. The linearity property also holds for the **RowMiss** constraint. Intuitively, since the size of the minimal automaton is directly correlated to the scalability, **RowHit** and **RowMiss** constraints are preferred for large problems.

D. Example

We now provide an example to illustrate how the automata differ between constraint types. In particular, we focus on **AnyHit** and **RowHit** constraints, that have been the subject of our theoretical investigation.

Example 1 (Weakly-Hard Automata)**.** *Given the two weakly-hard constraints $\lambda_1 = \binom{1}{3}$ and $\lambda_2 = \left\langle{2 \atop 6}\right\rangle$, we apply Theorems 1 and 2 and confirm that there is no partial ordering between the constraints, i.e. $\lambda_1 \npreceq \lambda_2$ and $\lambda_2 \npreceq \lambda_1$. Following the steps in Algorithm 1, we generate the* minimal *automaton representations of the two constraints, i.e., $\mathcal{G}_{\lambda_1}$ and $\mathcal{G}_{\lambda_2}$. The automaton representing the constraint set $\Lambda = \{\lambda_1, \lambda_2\}$, i.e., $\mathcal{G}_{\Lambda}$, is also generated and subsequently minimised. The results are shown in Figure 1, where the leftmost, middle, and rightmost automata correspond respectively to $\mathcal{G}_{\lambda_1}$, $\mathcal{G}_{\lambda_2}$, and $\mathcal{G}_{\Lambda}$.*

One of the most important novelties presented in this paper is the possibility to analyse weakly-hard constraint *sets* containing *all* the weakly-hard constraints types from Definition 3. Prior work proposed alternative solutions to the automaton generation problem, handling either a specific type of constraint [30], or a separate solution for each individual constraint type [20]. Our aim is to switch the focus to the applicability and scalability of the constraint representation, and hence substitute **AnyHit** and **AnyMiss** with **RowHit** and **RowMiss** whenever possible. Being able to analyse sets of constraints in a scalable way brings us one step closer to the analysis of real systems, in which window lengths are quite large. Additionally, for real systems it is often easier to constrain hits (e.g., via execution in a protected environment without interference) rather than the maximum number or the pattern of deadline misses.

E. *WeaklyHard.jl* functionality

The most relevant functions provided by **WeaklyHard.jl** are summarised in Table I.[3] In addition to the automata generation, the toolbox provides functions to compare constraints and obtain answers about their dominance and equivalence, to reduce a set of constraints to their dominant subset, and to generate sequences of arbitrary length satisfying sets of weakly-hard constraints. We also included a function that generate the satisfaction set $\mathcal{S}_N(\Lambda)$ from a graph $\mathcal{G}_{\Lambda}$. In addition to the functions presented in Table I, additional functions are included as syntactic sugar for a better user experience.

V. Experimental evaluation

We evaluate here the performance of **WeaklyHard.jl**.[4] First, we assess the scalability of the automaton generation, comparing **WeaklyHard.jl** with the state-of-the-art **WHRTgraph** [20], [21]. Then, we conduct a sensitivity analysis of **WeaklyHard.jl** to determine which parameters affect the execution time for the automata generation in cases that cannot be handled with other tools, e.g., sets of weakly-hard constraints. We provide results on how the type of constraints, maximum window length, and constraint set cardinality affect the computation time needed to generate the automaton. Finally, we investigate the average cardinality of the dominant set as a function of the cardinality of a set of constraints.

A. *Comparing* ***WeaklyHard.jl*** *and* ***WHRTgraph***

The literature contribution that is closest to our research is **WHRTgraph** [20], [21]. **WHRTgraph**'s analysis of weakly-hard tasks is also based on the construction of automata. While **WHRTgraph** handles only one weakly-hard constraint at a time, it can construct the automaton that correspond to **AnyHit** and **RowHit** constraints, making it the reference in

[3]The code includes a README file that guides the user through the setup of the package and provides simple usage examples. The only prerequisite is the Julia interpreter and compiler, available at https://julialang.org.

[4]All the reported experiments ran on an Intel Xeon E5-2620 v3 @ 2.40GHz CPU with 126GB RAM memory.

TABLE I
FUNCTIONS OFFERED BY **WeaklyHard.jl**.

Function	**Description**
`AnyHitConstraint(x, k)`	Defines a constraint $\lambda = \binom{x}{k}$
`AnyMissConstraint(x, k)`	Defines a constraint $\lambda = \overline{\binom{x}{k}}$
`RowHitConstraint(x, k)`	Defines a constraint $\lambda = \left\langle{x \atop k}\right\rangle$
`RowMissConstraint(x)`	Defines a constraint $\lambda = \overline{\langle x \rangle}$
`is_satisfied(Lambda, w)`	Returns **true** if $w \vdash \Lambda$, i.e., if the word w satisfies all the constraints in Λ, and **false** otherwise (note: can be invoked also passing a single constraint λ as parameter)
`is_dominant(lambda1, lambda2)`	Returns **true** if $\lambda_1 \preceq \lambda_2$ and **false** otherwise
`is_equivalent(lambda1, lambda2)`	Returns **true** if $\lambda_1 \equiv \lambda_2$ and **false** otherwise
`dominant_set(Lambda)`	Returns $\Lambda^* \subseteq \Lambda$
`build_automaton(Lambda)`	Returns the automaton $\mathcal{G}_\Lambda$ (note: can be invoked also passing a single constraint λ as parameter)
`minimize_automaton!(G)`	Returns the minimal representation of $\mathcal{G}_\Lambda$ (note: changes $\mathcal{G}_\Lambda$)
`random_sequence(G, N)`	Returns a word w, $\lvert w\rvert = N$ obtained through an N-step random walk in $\mathcal{G}_\Lambda$
`all_sequences(G, N)`	Returns the satisfaction set $\mathcal{S}_N(\Lambda)$ corresponding to $\mathcal{G}_\Lambda$

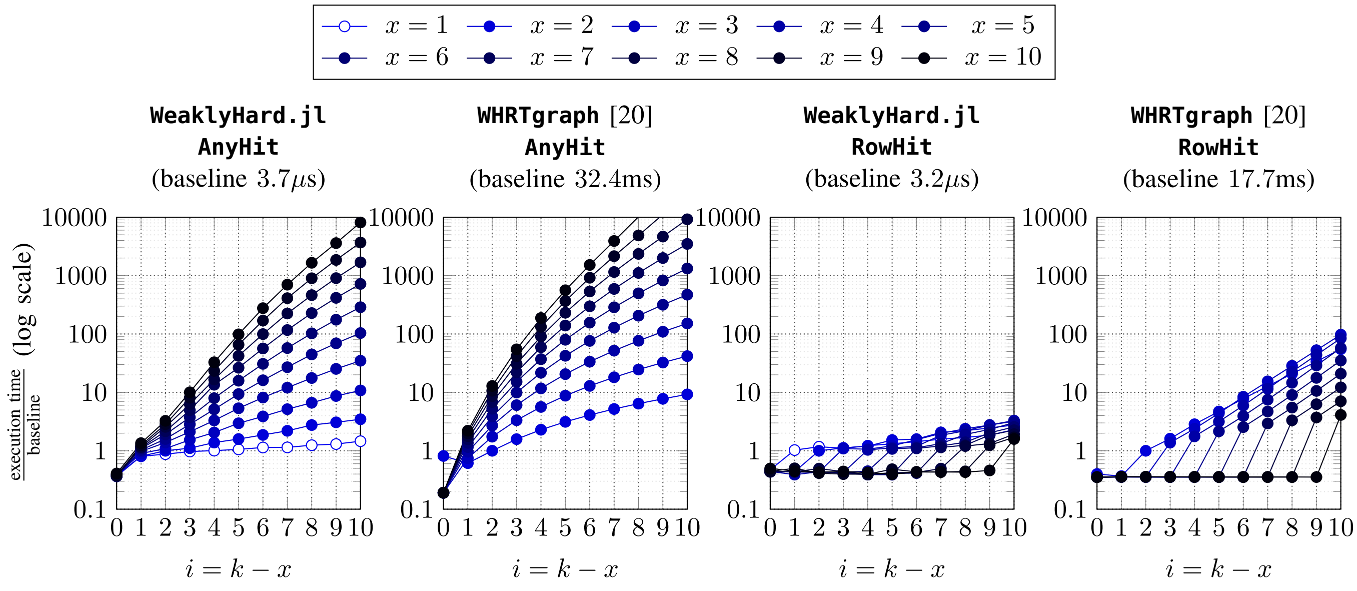


Fig. 2. Execution time comparison for **AnyHit** and **RowHit** constraints with **WeaklyHard.jl** and **WHRTgraph** [20] increasing the difference between window size and number of hits constrained. Baseline values are reported on top of the corresponding plots.

terms of analysis capabilities. **WHRTgraph** is implemented in MATLAB, while **WeaklyHard.jl** is implemented in Julia. Hence, comparing the execution times of the two (on their own) is pointless. Furthermore, we are more interested in assessing the scalability to an increase in the constraint window size than the absolute numbers for the execution times. We therefore define a baseline case, for a fair comparison, i.e., the reported results are fractions and multiples of the baseline, which is different for each tool and constraint type.

To test the scalability of the automaton generation, we ask both **WeaklyHard.jl** and **WHRTgraph** to generate the automata that correspond to the **AnyHit** $\binom{x}{k}$ and **RowHit** $\left\langle{x \atop k}\right\rangle$ constraints for $x \in \{1, 2, \ldots, 10\}$, $k = x + i$ and $i \in \{0, 1, \ldots, 10\}$. We divide the obtained results by the baseline value, i.e., the execution time needed for the corresponding tool to generate the automaton for the given constraint type, $x = 2$ and $k = 4$.[5]

Figure 2 shows the mean value of the execution time for the automaton generation, divided by the corresponding baseline value, using a logarithmic y-axis. The baseline computation times for **AnyHit** constraint are 3.7μs for **WeaklyHard.jl** and 32.4ms for **WHRTgraph**. On the contrary, for a **RowHit** constraint, the baseline computation time is 3.2μs for **WeaklyHard.jl** and 17.7ms for **WHRTgraph**. Due

[5]The choice of the baseline case reflects the simplest constraint that is correctly handled by both **WeaklyHard.jl** and **WHRTgraph**. Comparing the methods, we unveiled that **WHRTgraph** is unable to find an automaton for constraints in which $x = 1$. The two plots for **WHRTgraph** in Figure 2 do not contain results for $x = 1$ (white filled markers) precisely due to this problem.

to the extensive computational time necessary to build the automata using **WHRTgraph**, each automaton was built 30 times (i.e., each point in the figure is the mean of 30 executions). **WeaklyHard.jl** is significantly faster, thus, each automata was built 100 000 times to reduce the execution time variance.

WHRTgraph represents a weakly-hard constraint with a slightly different, yet equivalent automaton to the one generated by **WeaklyHard.jl**. In particular, the automaton generated by **WHRTgraph** has fewer vertices and weights on the edges encode the number of consecutive deadline misses allowed between the vertices. Thus, a transition between two vertices in **WHRTgraph** is not equivalent to one outcome (as for **WeaklyHard.jl**), reducing flexibility, i.e., making it harder for example to automatically generate code to monitor the outcomes of task executions. Multiple successive outcomes for each transition also complicate the handling of sets of weakly-hard constraints. In terms of scalability, an automaton representation with fewer nodes may sound more efficient. However, we show that **WeaklyHard.jl** scales better than **WHRTgraph** by more than an order of magnitude. The baseline numbers show that **WeaklyHard.jl** is also significantly faster in absolute terms.

Comparing the scalability of the two tools for **AnyHit** constraints (leftmost plots), we observe that **WeaklyHard.jl** is more than an order of magnitude faster than **WHRTgraph**. On the contrary, for **RowHit** constraints (rightmost plots), we experience a speedup of almost two orders of magnitude for high values of $i = k - x$. The scalability of the **RowHit** constraints are further investigated in the following subsection.

B. *Evaluating **RowHit** constraints*

In the previous subsection we discussed the scalability of **WeaklyHard.jl** compared to the state-of-the-art. Despite improvements of more than an order of magnitude (not considering the baseline), the time necessary to construct the automata for **AnyHit** constraints grows rapidly with increasing window lengths. Motivated by the ongoing discussion on the practical importance of consecutive deadline hits [2], [33] and the scalability considerations presented in Section IV-C, we now perform an extensive evaluation of the scalability of the **RowHit** constraints.

Using **WeaklyHard.jl**, we generate the automaton corresponding to the **RowHit** $\left\langle {x \atop k} \right\rangle$ constraints for $x \in \{1, 2, \ldots, 15\}$, $k \in \{x, x+1, \ldots, 100\}$. To the best of our knowledge, this is the first research work that generates automata representations of weakly-hard constraints with window lengths above 100. Figure 3 displays the mean execution time over 100 executions for the automata generation using a logarithmic scale, showing a piecewise exponential growth of execution time with some jumps. Despite having constraints with window lengths up to $k = 100$, the worst reported execution time is below 7 seconds; reinforcing the arguments in favour of using **RowHit** rather than **AnyHit** constraints.

Another interesting consideration is related to the jumps in the execution time that each line shows when reaching certain values of x and k. This follows from the choice of using

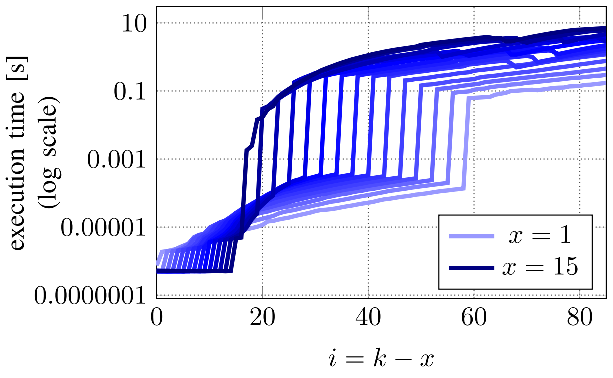


Fig. 3. Mean execution time of the generation **RowHit** constraint automaton.

integers to represent words in **WeaklyHard.jl**. For constraints where $2x + k \geq 64$, 64 bit integers are not enough to represent all sequences, and **WeaklyHard.jl** consequently converts the sequence representation to big integers (using more than 64 bits). This representation requires additional resources (memory and computation), hence producing execution time jumps.

C. *Analysing sets of weakly-hard constraints*

WeaklyHard.jl is the first tool that provides the ability to analyse sets of weakly-hard constraints. In the following we conduct a sensitivity analysis to assess the scalability of the automaton generation for a set of weakly hard constraints. In particular, we are interested in finding how the window size affects the execution time of the tool, and how the composition of the set influences the execution time.

We randomise dominant sets of constraints, imposing that at least one of the constraints has a window size of $k_{\max} \in \{10, 11, \ldots, 30\}$. We generate sets with either $|\Lambda^*| = 2$ or $|\Lambda^*| = 4$. We allow these sets to include one **RowHit** constraint or none. The results of our study are shown in Figure 4. For each of the values of $k_{\max}$ in the figure, we generate 50 dominant sets Λ^*. The figure shows the average execution time in seconds (as a line) and the area representing the span between minimum and maximum execution time.

The first conclusion that we can draw is that the average execution times follow straight lines in a logarithmic scale, thus clearly pointing to the exponential time complexity inherent to expressive task models, such as the weakly-hard model [27].

When the cardinality of the set $|\Lambda^*|$ increases (i.e., comparing the two leftmost and the two rightmost plots) the maximum execution time does not change significantly. In fact, states that would have been reachable with fewer constraint become unreachable due to the additional constraints pruning the state-space. However, we experience a slight reduction in the execution time's variance, which follows from the nature of the dominant set. Comparing two dominant sets, Λ_1^* and Λ_2^*, with the same $k_{\max}$: when $|\Lambda_1^*| = 2$ and $|\Lambda_2^*| = 4$, the set Λ_2^* must include less restrictive constraints (otherwise they would dominate the other constraints in the set). Hence, the set Λ_2^* is less likely to be trivial to analyse.

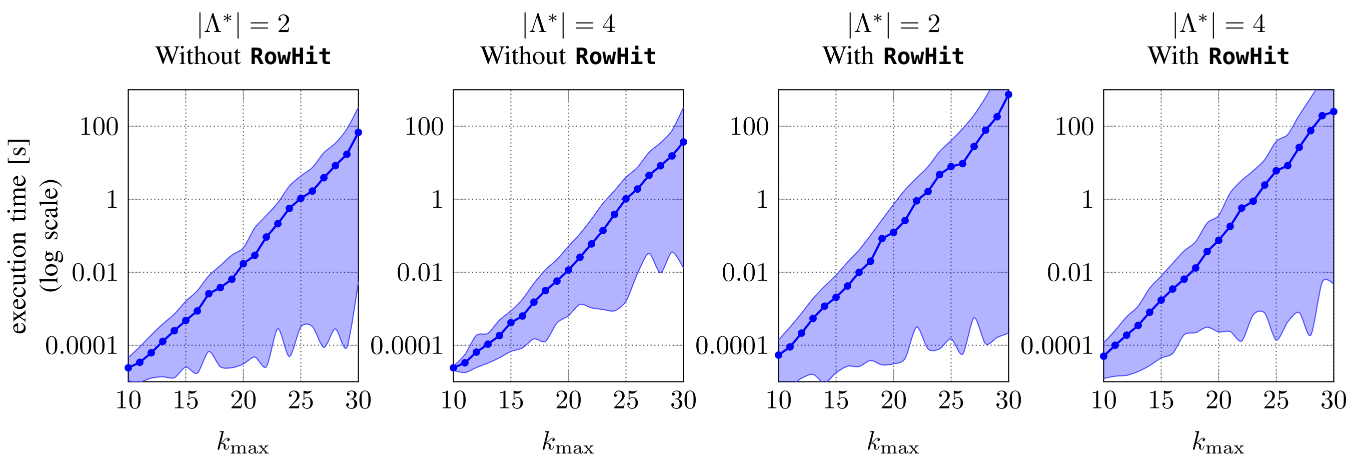


Fig. 4. Execution time comparison for the generation of the automaton for sets of constraints with increasing maximum window sizes $\max k$. Average values are reported alongside the areas between minimum and maximum execution times.

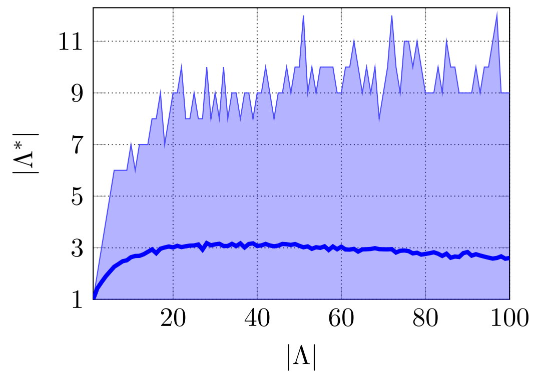


Fig. 5. Average cardinality of the dominant set Λ^* as a function of $|\Lambda|$ with $k_{\max} = 100$ for 1000 randomly generated constraint sets Λ.

Finally, including a **RowHit** constraint in the set Λ^* increases the execution time by an order of magnitude. This follows from the complex interconnections between the **RowHit** and remaining weakly-hard constraints. Particularly, for the **AnyHit**, **AnyMiss**, and **RowMiss** constraints it is sufficient to count the deadline hits of the jobs currently in the window; however, the **RowHit** constraints need to keep additional track of when they appeared. This is further reinforced by the fact that when a dominant set includes a **RowHit** constraint, the other constraints in the set have to be very conservative in order to neither dominate nor be dominated by it. However, we remark that **WeaklyHard.jl** is able to generate an automaton for a set Λ^* of 4 constraints with $k_{\max} = 30$, including a **RowHit** constraint, in less than 200 seconds.

D. Determining the dominant constraint set

In Section V-C we investigated dominant sets Λ^* with cardinality $|\Lambda^*| \in \{2, 4\}$. Here we justify why this is a relevant benchmark despite the low cardinality.

We select a maximum window size $k_{\max} = 100$. The window size is large enough that we can find an expressive variety of constraints without partial ordering. We randomly generate sets Λ containing $|\Lambda| \in \{1, \ldots, 100\}$ constraints. For each value of $|\Lambda|$ we generate 1000 different sets, excluding all the trivial constraints that would reduce to $\overline{\lambda}$ and $\underline{\lambda}$. We then compute the dominant set Λ^* corresponding to each set. Figure 5 shows the average cardinality of Λ^* (solid line) and the experienced range (area).

As can be seen, most constraint sets reduce to dominant sets with cardinality less than 4, thus motivating our investigation of the automaton generation execution time. Generally, it is also interesting that additional constraints tends to reduce the cardinality of Λ^*, after a peak is reached. This is however not surprising seeing as adding constraints increases the chances of the added constraints being dominant over some of the constraints in the set.

VI. Conclusion

The research behind this paper is motivated by the attention the weakly-hard model is receiving in both academic and industrial contexts. The paper primarily proposes two contributions: (i) two novel theorems that complete the relation graph between weakly-hard constraints of different types, and (ii) an open-source tool, **WeaklyHard.jl**, that helps in the analysis of weakly-hard tasks. The tool includes functions to relate different weakly-hard constraints to one another, and functions to generate automata that encode the feasible outcomes of weakly-hard tasks.

We envision **WeaklyHard.jl** to be used for (i) the analysis of complex tasksets, in which tasks are subject to different weakly-hard constraints, possibly with large windows, (ii) the generation of monitoring code that provides runtime checks for the satisfaction of weakly-hard constraints. As an example, to validate the conjectures that became the theorems of Section III-A, we used **WeaklyHard.jl** to generate the satisfaction sets for various pairs of **AnyHit** and **RowHit** constraints. We then calculated the intersection between the generated sets to verify that our conjecture held for the specific cases under test.

We analyse the scalability of **WeaklyHard.jl** and the dominance between different constraints. Furthermore, we build dominant sets of constraints. To the best of our knowledge, **WeaklyHard.jl** is the first tool that enables the analysis of tasks that satisfy sets of weakly-hard constraints.

References

[1] L. Ahrendts, S. Quinton, T. Boroske, and R. Ernst. Verifying weakly-hard real-time properties of traffic streams in switches networks. In S. Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106, pages 15:1–15:22, 2018.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[3] A. Behrouzian, H. Ara, M. Geilen, D. Goswami, and T. Basten. Firmness analysis of real-time tasks. *ACM Trans. Embed. Comput. Syst.*, 19(4), July 2020.

[4] G. Bernat. *Specification and analysis of weakly hard real-time systems*. PhD thesis, Department de les Ciéncies Matemátiques i Informática, Universitat de les Illes Balears, Spain, 1998.

[5] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50:308 – 321, 2001.

[6] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[7] A. Burns and R. Davis. Mixed criticality systems – a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[8] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems*. Springer, 2005.

[9] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg. Response-time analysis of ROS2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133, pages 6:1–6:23, 2019.

[10] E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computing*, 205(8):1149–1172, 2007.

[11] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–82, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[12] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[13] Z. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 584–589, 2017.

[14] Z. A. H. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 584–589, 2017.

[15] Z. A. H. Hammadeh, S. Quinton, M. Panunzio, R. Henia, L. Rioux, and R. Ernst. Budgeting under-specified tasks for weakly-hard real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

[16] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[17] C. Huang, W. Li, and Q. Zhu. Formal verification of weakly-hard systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 197–207, New York, NY, USA, 2019. Association for Computing Machinery.

[18] C. Huang, K. Wardega, W. Li, and Q. Zhu. Exploring weakly-hard paradigm for networked systems. In *Proceedings of the Workshop on Design Automation for CPS and IoT*, DESTION '19, page 51–59, New York, NY, USA, 2019. Association for Computing Machinery.

[19] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 110–117, 1995.

[20] S. Linsenmayer and F. Allgöwer. Stabilization of networked control systems with weakly hard real-time dropout description. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4765–4770, 2017.

[21] S. Linsenmayer, M. Hertneck, and F. Allgöwer. Linear weakly hard real-time control systems: Time- and event-triggered stabilization. *IEEE Transactions on Automatic Control*, 66(4):1932–1939, 2021.

[22] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein. Control-system stability under consecutive deadline misses constraints. In M. Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[23] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *ACM Trans. Embed. Comput. Syst.*, 3(4):706–735, Nov. 2004.

[24] P. Pazzaglia, L. Pannocchi, A. Biondi, and M. Di Natale. Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses. In S. Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] P. Pazzaglia, Y. Sun, and M. Di Natale. Generalized weakly hard schedulability analysis for real-time periodic tasks. *ACM Trans. Embed. Comput. Syst.*, 20(1):3:1–3:26, 2021.

[26] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

[27] M. Stigge and W. Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 51:602–636, 2015.

[28] Y. Sun and M. Di Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Trans. Embed. Comput. Syst.*, 16(5s):171:1–171:19, 2017.

[29] G. Tu, J.-l. Li, F.-m. Yang, and W. Luo. Relationships between window-based real-time constraints. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 394–399, 2007.

[30] E. van Horssen, A. Behrouzian, D. Goswami, D. Antunes, T. Basten, and W. Heemels. Performance analysis and controller improvement for linear systems with (m, k)-firm data losses. In *2016 European Control Conference (ECC)*, pages 2571–2577, 2016.

[31] M. van Osch and S. Smolka. Finite-state analysis of the can bus protocol. In *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*, pages 42–52, 2001.

[32] G. von der Brüggen, N. Piatkowski, K.-H. Chen, J.-J. Chen, and K. Morik. Efficiently approximating the probability of deadline misses in real-time systems. In S. Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[33] N. Vreman, A. Cervin, and M. Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In B. B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[34] S.-L. Wu, C.-Y. Bai, K.-C. Chang, Y.-T. Hsieh, C. Huang, C.-W. Lin, E. Kang, and Q. Zhu. Efficient system verification with multiple weakly-hard constraints for runtime monitoring. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 497–516, Cham, 2020. Springer International Publishing.

[35] W. Xu, Z. Hammadeh, A. Kröller, R. Ernst, and S. Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *27th Euromicro Conference on Real-Time Systems*, pages 247–256, 2015.

[36] H. Zeng and M. Di Natale. Schedulability analysis of periodic tasks implementing synchronous finite state machines. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 353–362, 2012.