Workload Models

Pontus Ekberg

Uppsala University

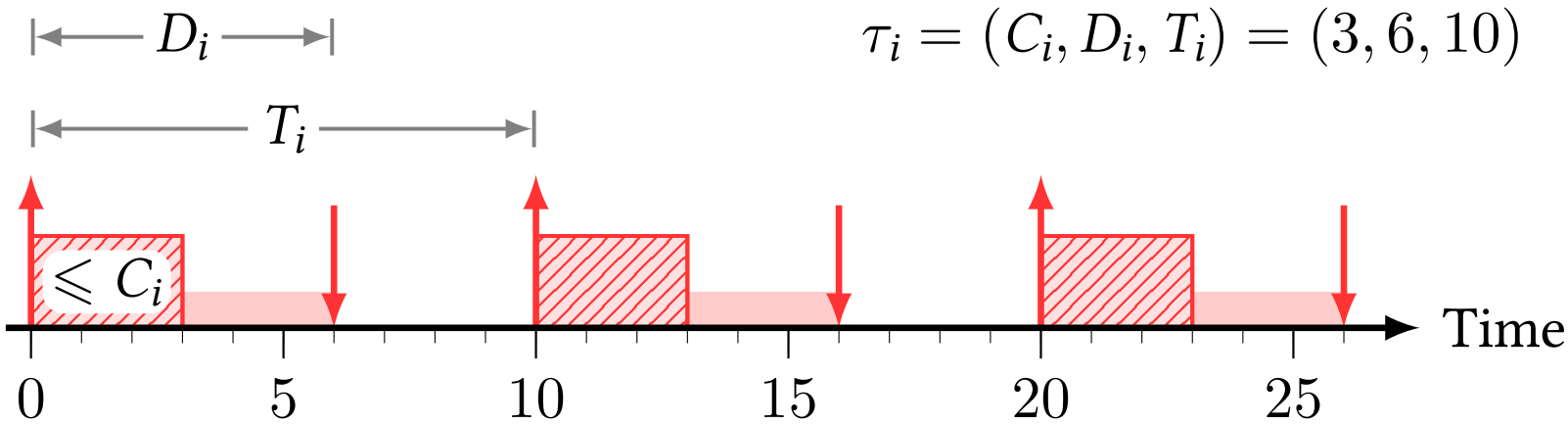
2018-10-11

(With some slides borrowed from Martin Stigge)

Recap

A sporadic task τ_i is given by a triple $(C_i, D_i, T_i) \in \mathbb{N}^3$, where

- C_i is the worst-case execution time (WCET),
- D_i is the relative deadline, and
- T_i is the minimum inter-release separation time (or just *period*).



Today's topic

1. We will look at a few *generalizations* of the sporadic task model, which allow for more complicated patterns of job releases.
2. We will outline analysis techniques for feasibility (or EDF-schedulability) for one of these task models.

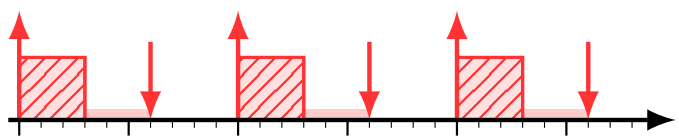
Recall the feasibility test for sporadic tasks

Feasibility test

A sporadic task set $\mathcal{T}$ is feasible on a single preemptive processor iff

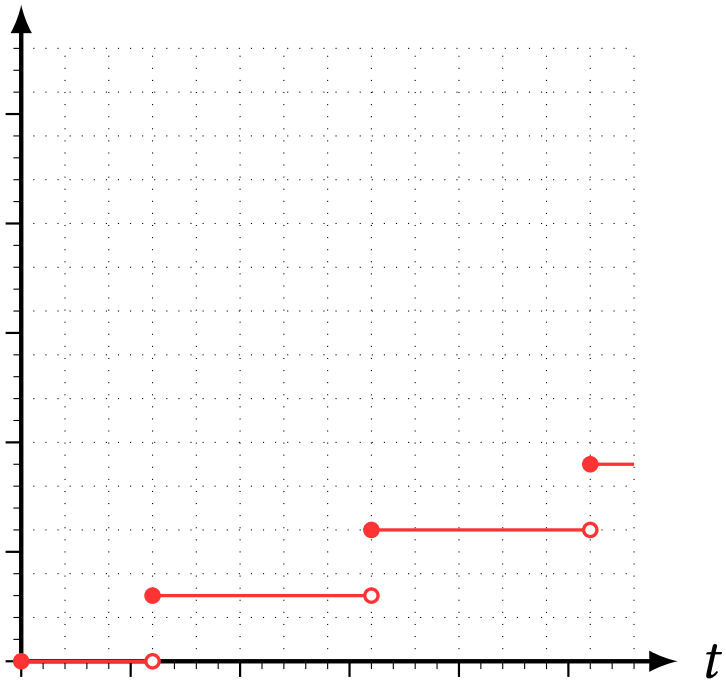
$$\forall t, \text{ such that } t \geqslant 0 : \quad \text{dbf}(\mathcal{T}, t) \leqslant t.$$

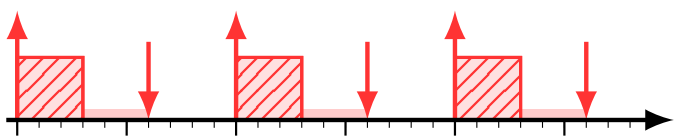
Recall the feasibility test for sporadic tasks



(C_i, D_i, T_i)

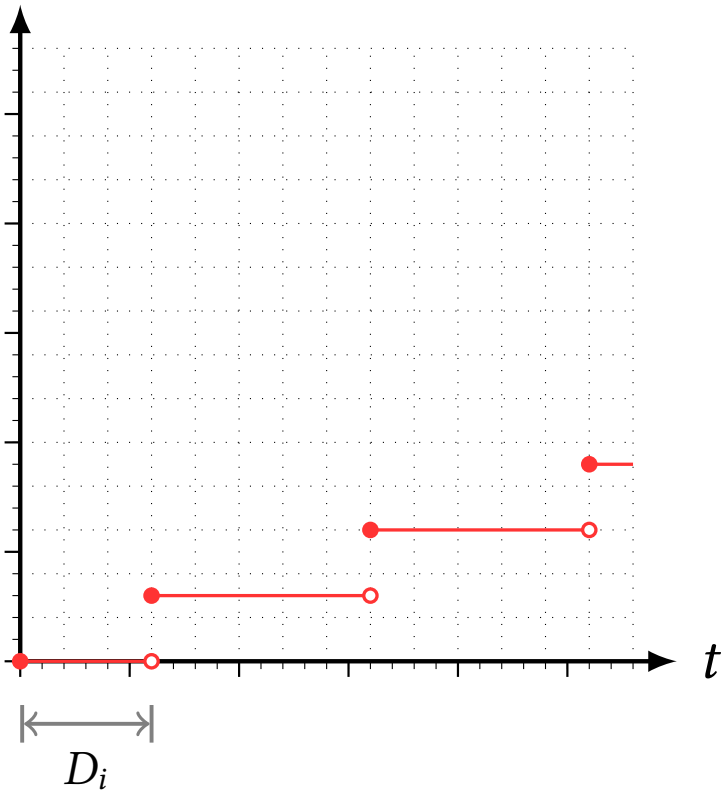
Recall the feasibility test for sporadic tasks

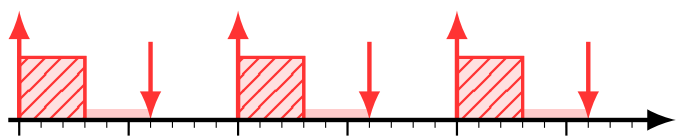




(C_i, D_i, T_i)

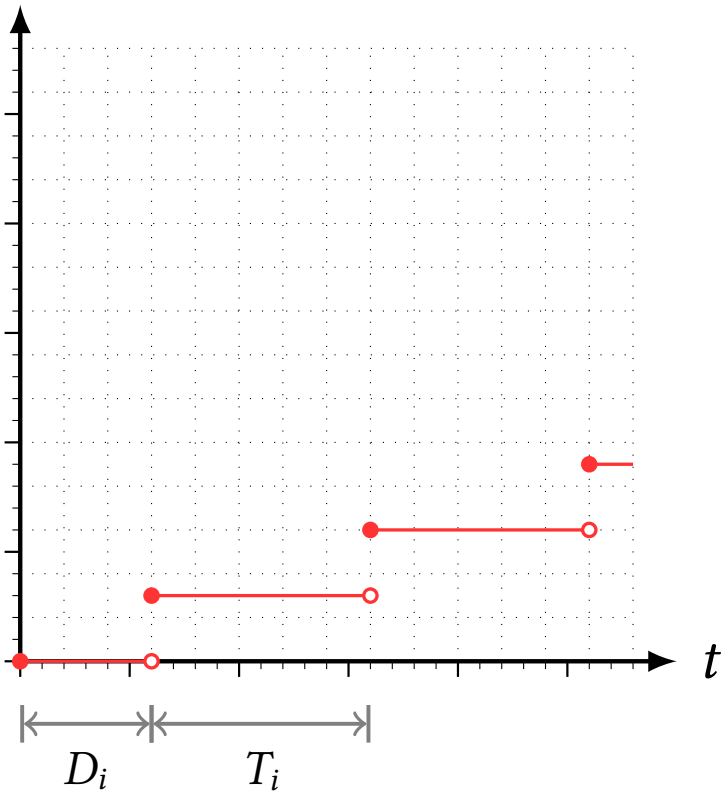
Recall the feasibility test for sporadic tasks

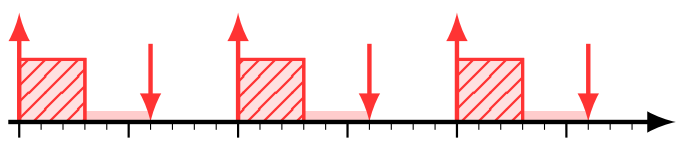




(C_i, D_i, T_i)

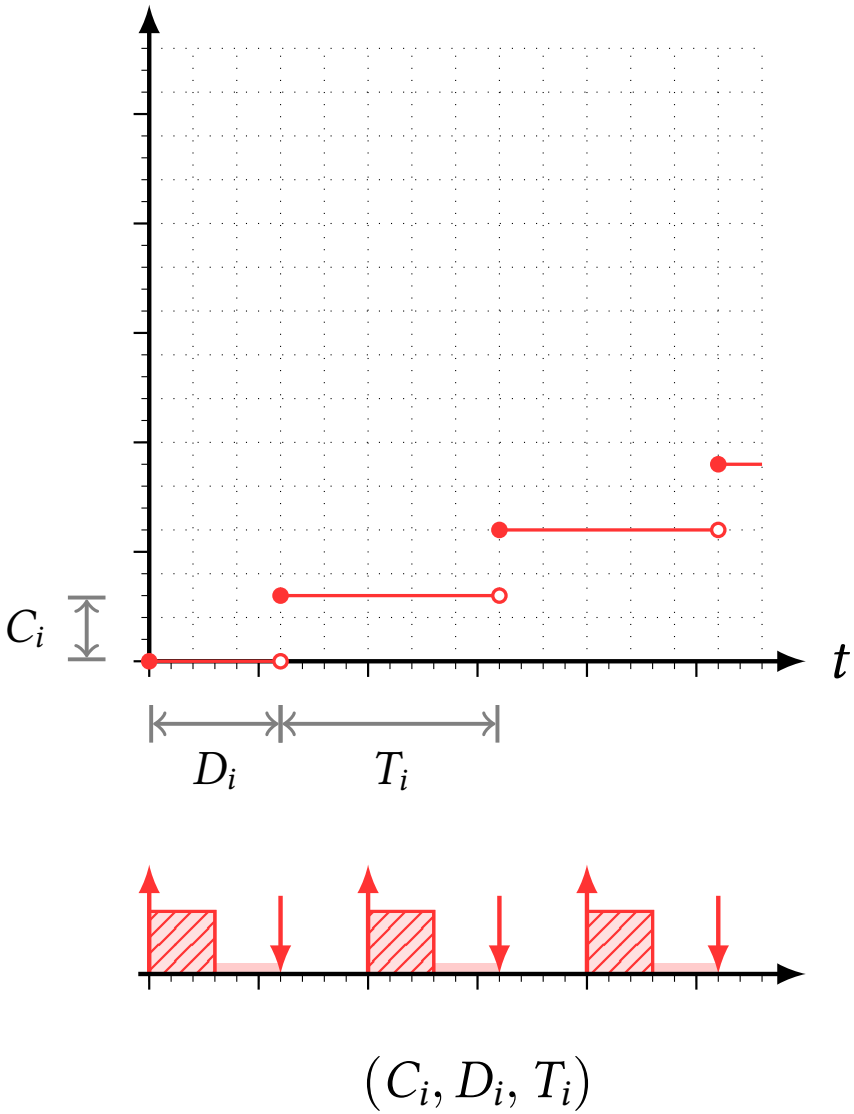
Recall the feasibility test for sporadic tasks





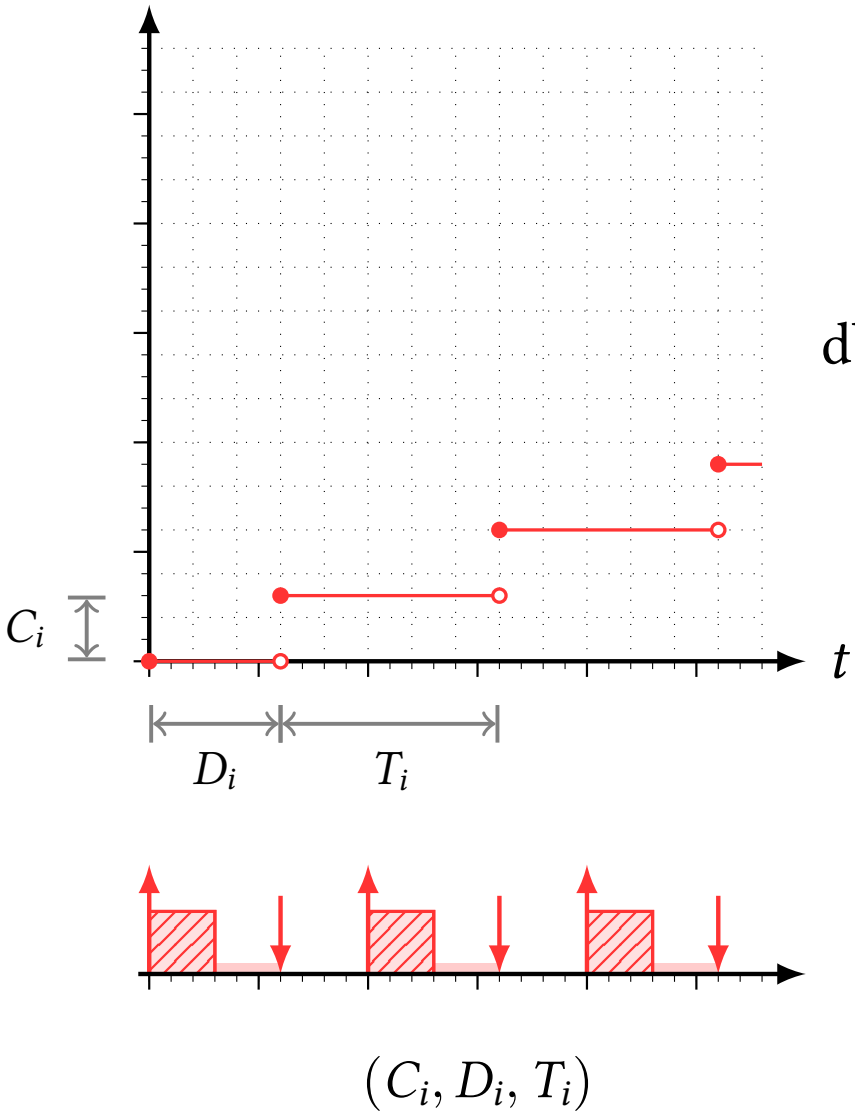
(C_i, D_i, T_i)

Recall the feasibility test for sporadic tasks



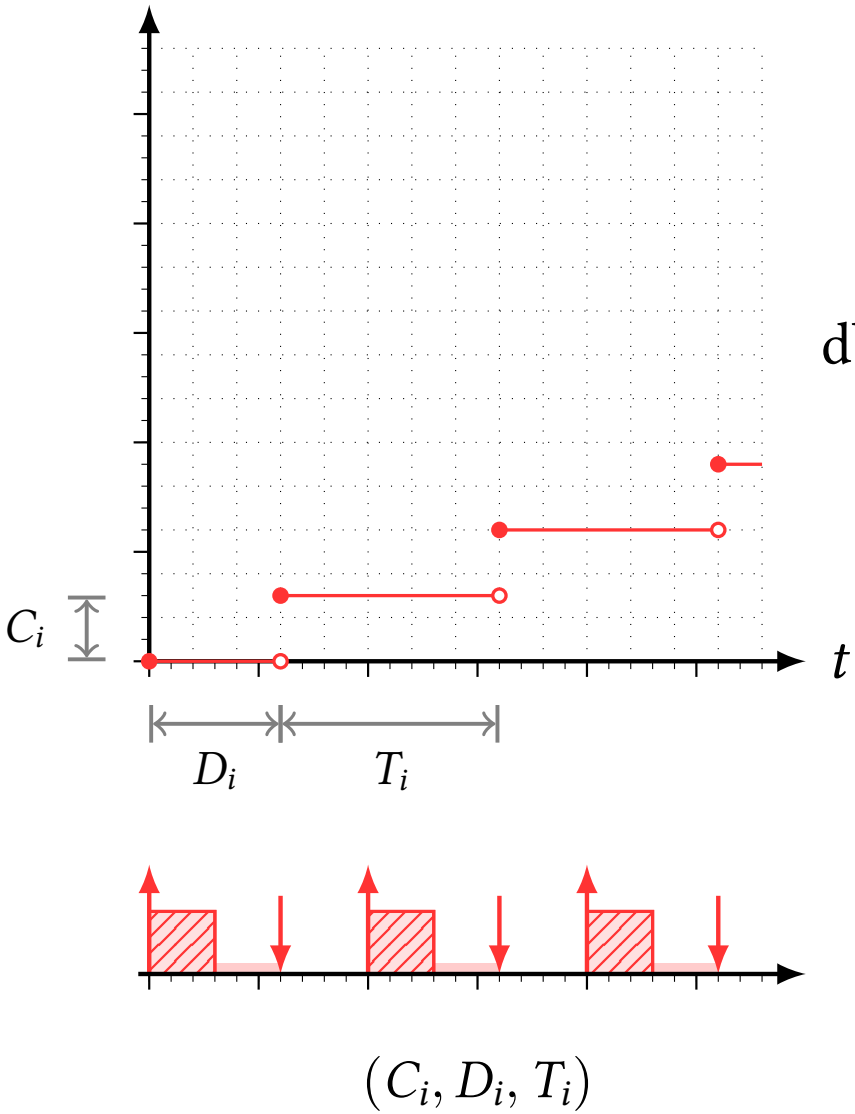
(C_i, D_i, T_i)

Recall the feasibility test for sporadic tasks



$$\mathrm{dbf}(\tau_i, t) = \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i$$

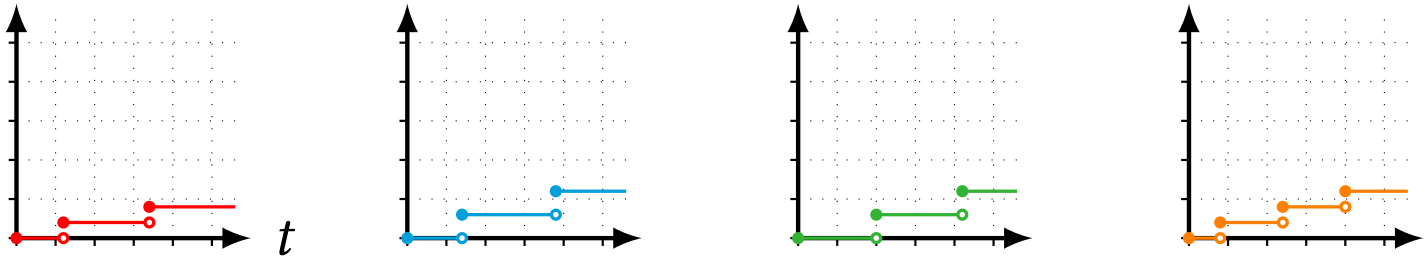
Recall the feasibility test for sporadic tasks



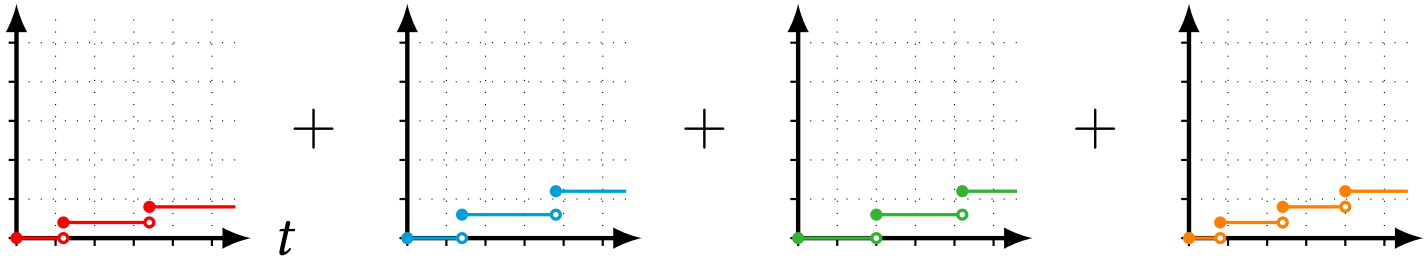
$$\mathrm{dbf}(\tau_i, t) = \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i$$

$$\mathrm{dbf}(\mathcal{T}, t) = \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}(\tau_i, t)$$

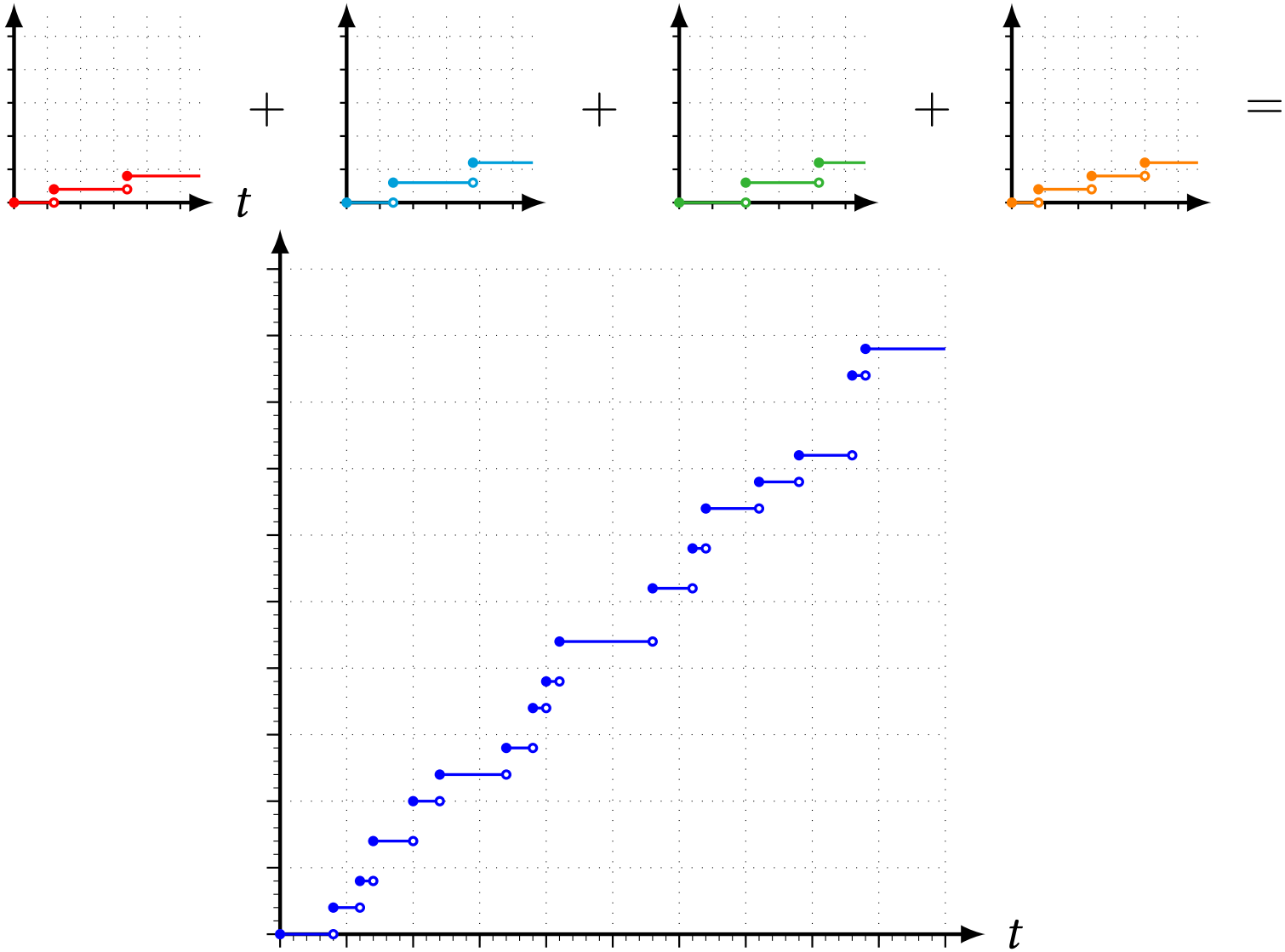
Recall the feasibility test for sporadic tasks



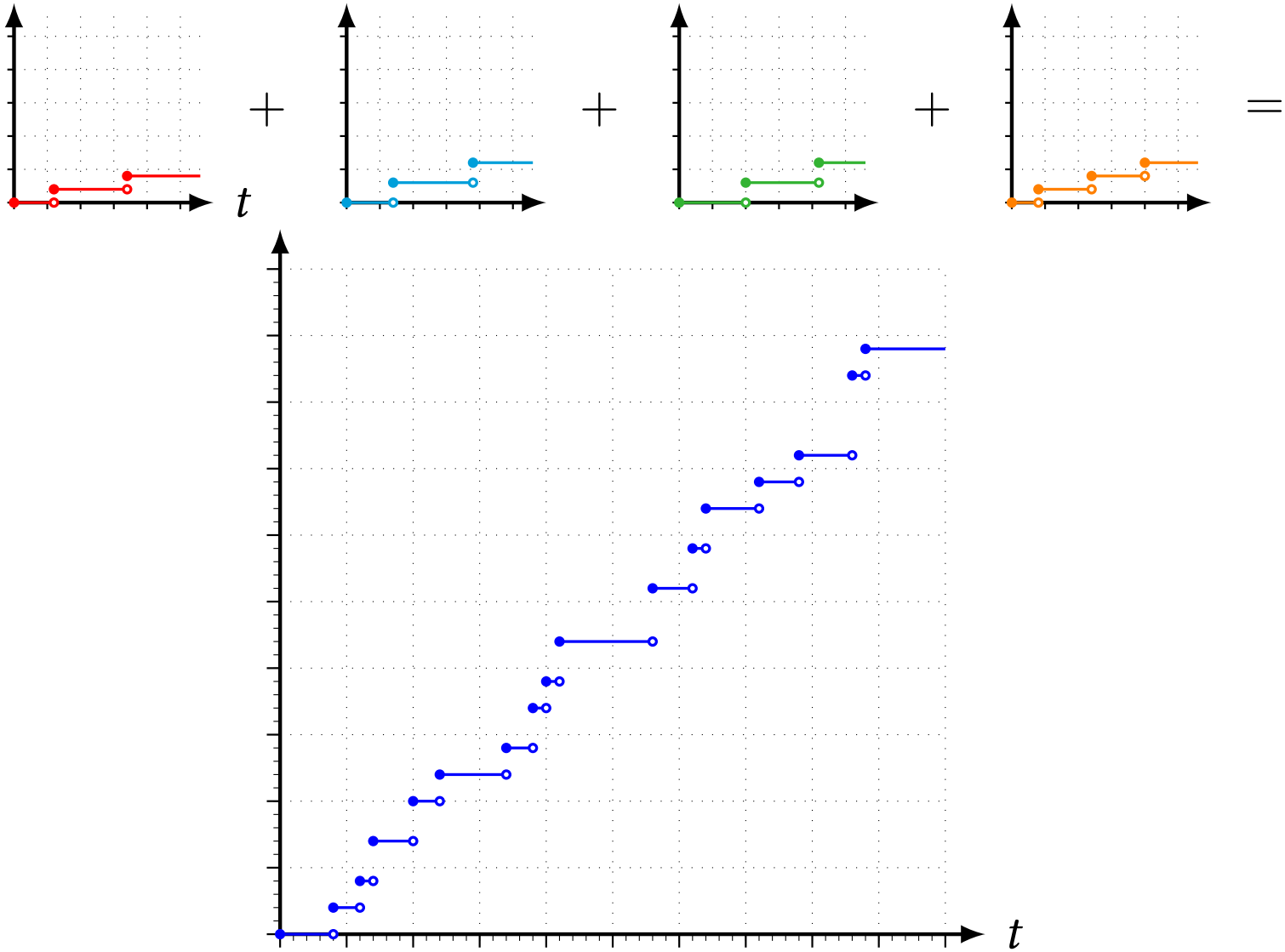
Recall the feasibility test for sporadic tasks



Recall the feasibility test for sporadic tasks

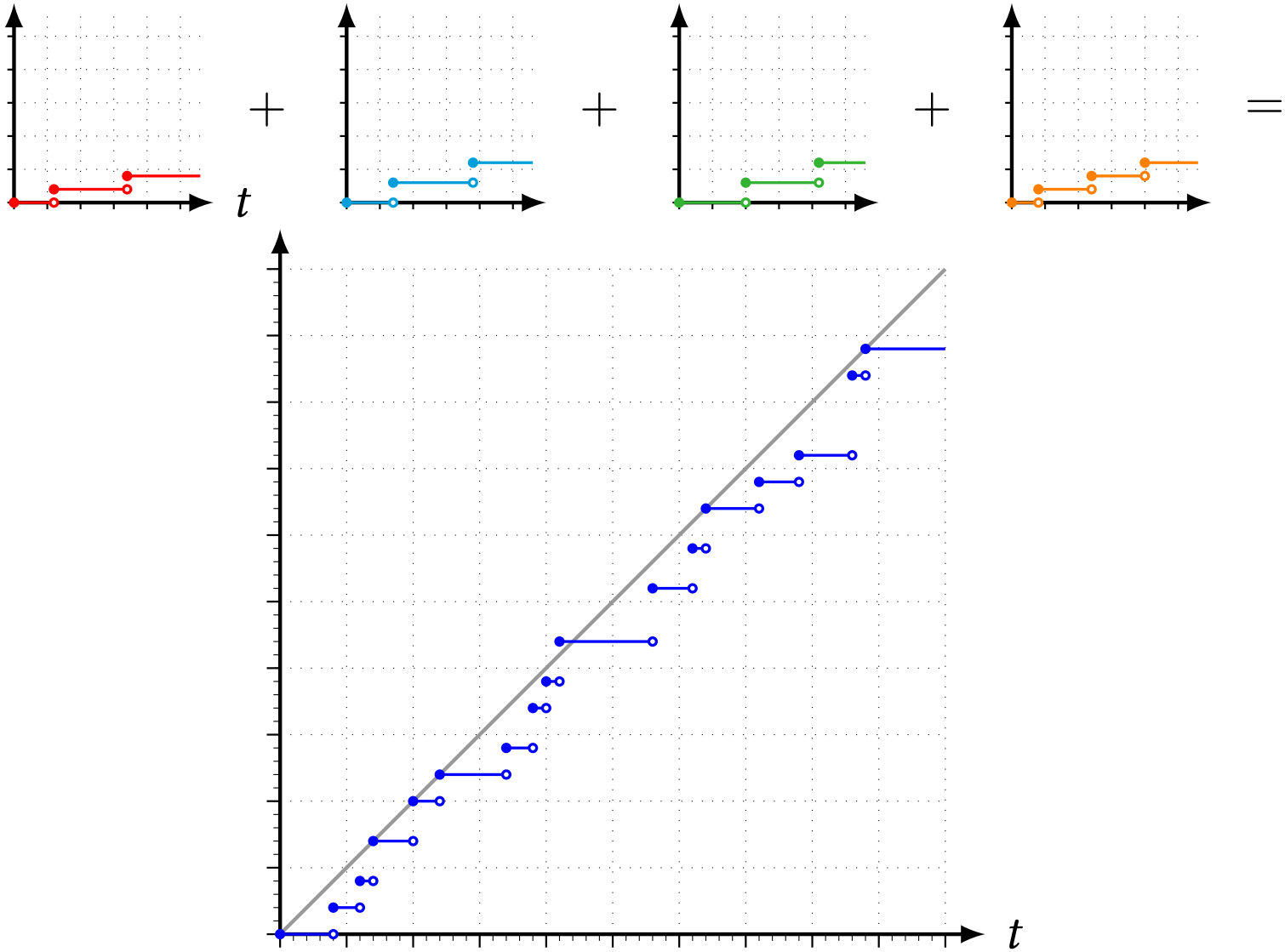


Recall the feasibility test for sporadic tasks



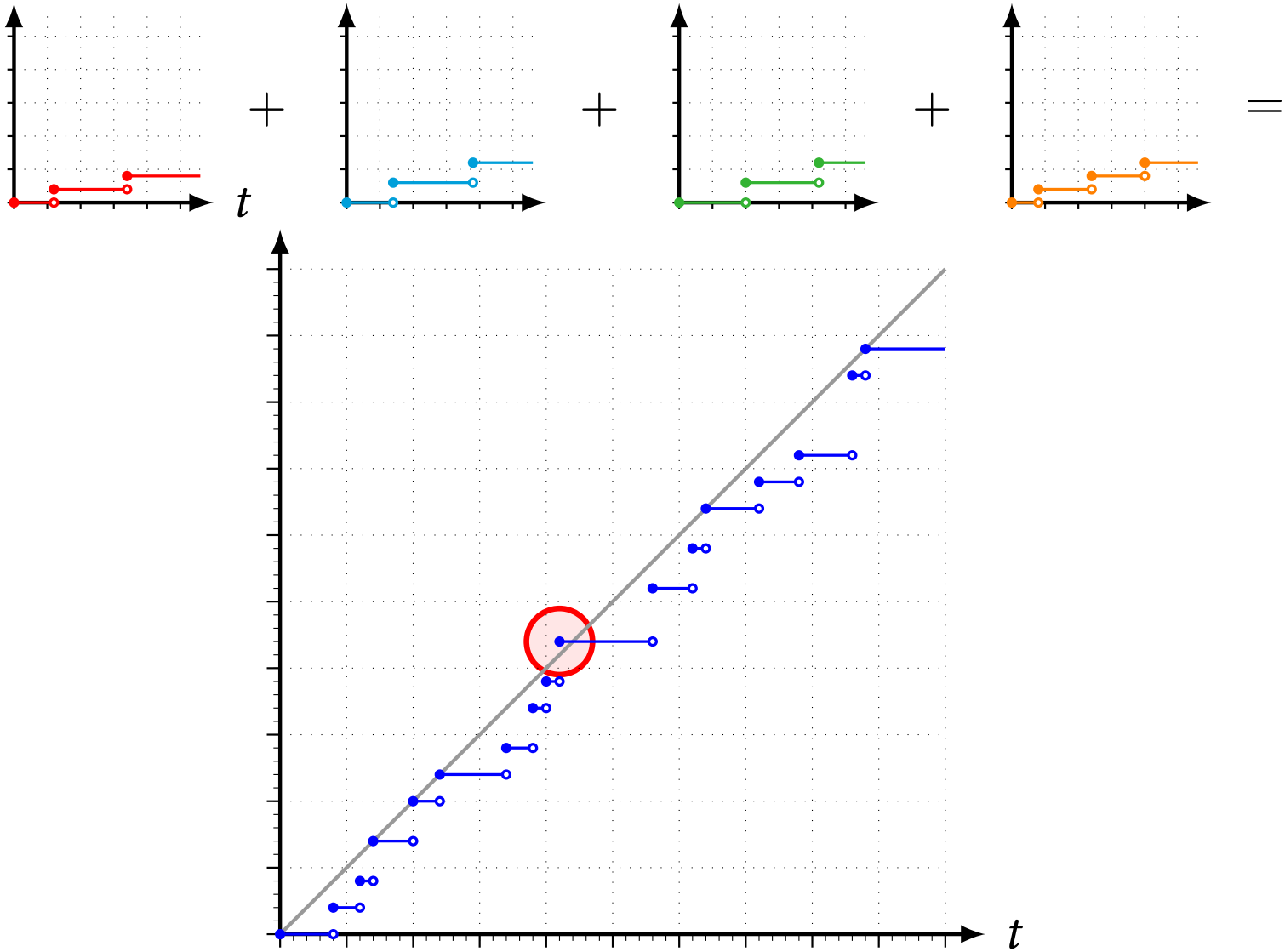
$$\forall t, \text{ such that } t \geqslant 0 : \quad \text{dbf}(\mathcal{T}, t) \leqslant t$$

Recall the feasibility test for sporadic tasks



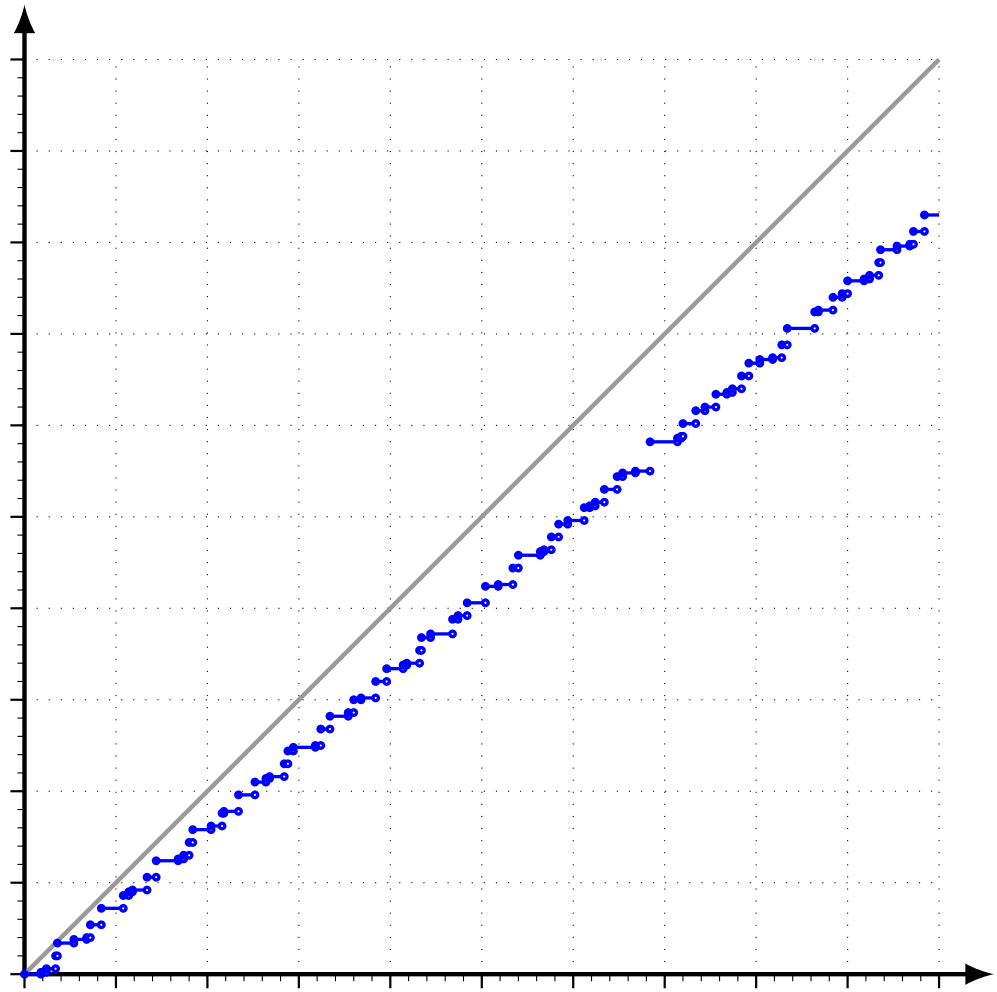
$$\forall t, \text{ such that } t \geqslant 0 : \quad \text{dbf}(\mathcal{T}, t) \leqslant t$$

Recall the feasibility test for sporadic tasks

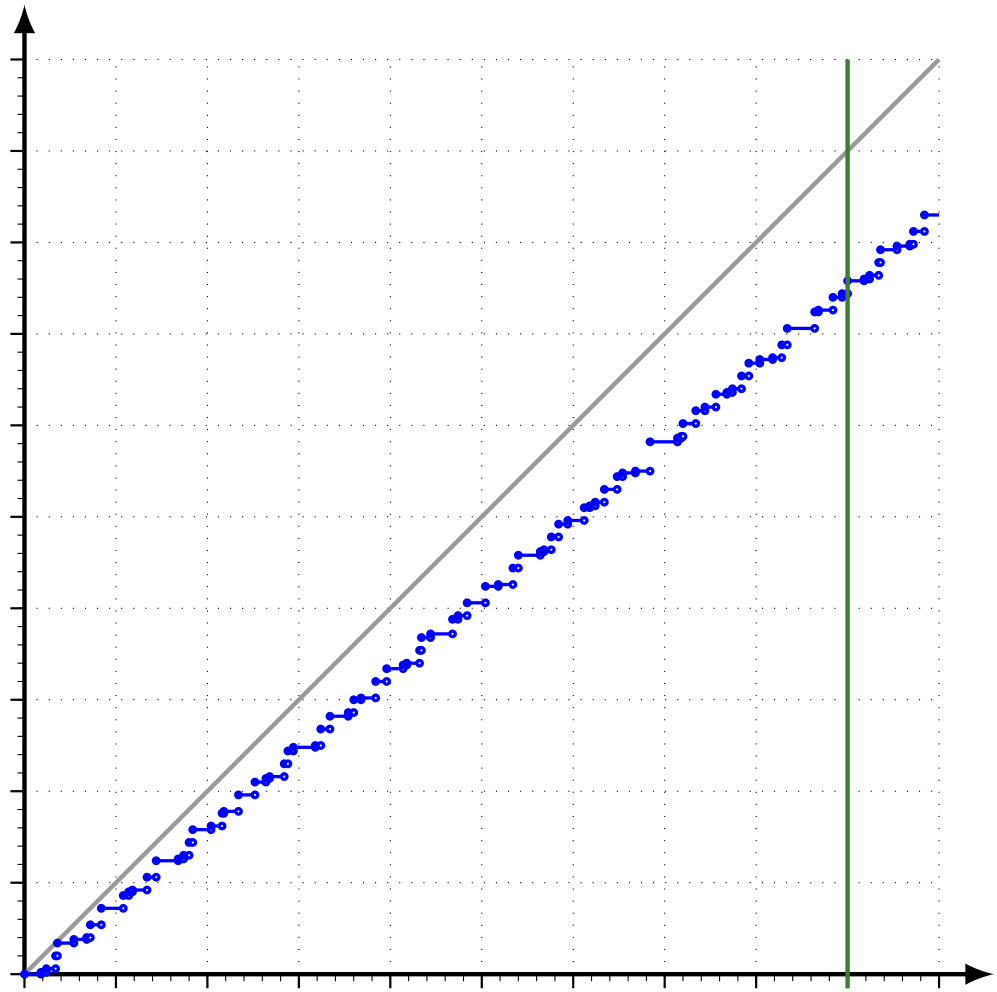


$$\forall t, \text{ such that } t \geqslant 0 : \quad \text{dbf}(\mathcal{T}, t) \leqslant t$$

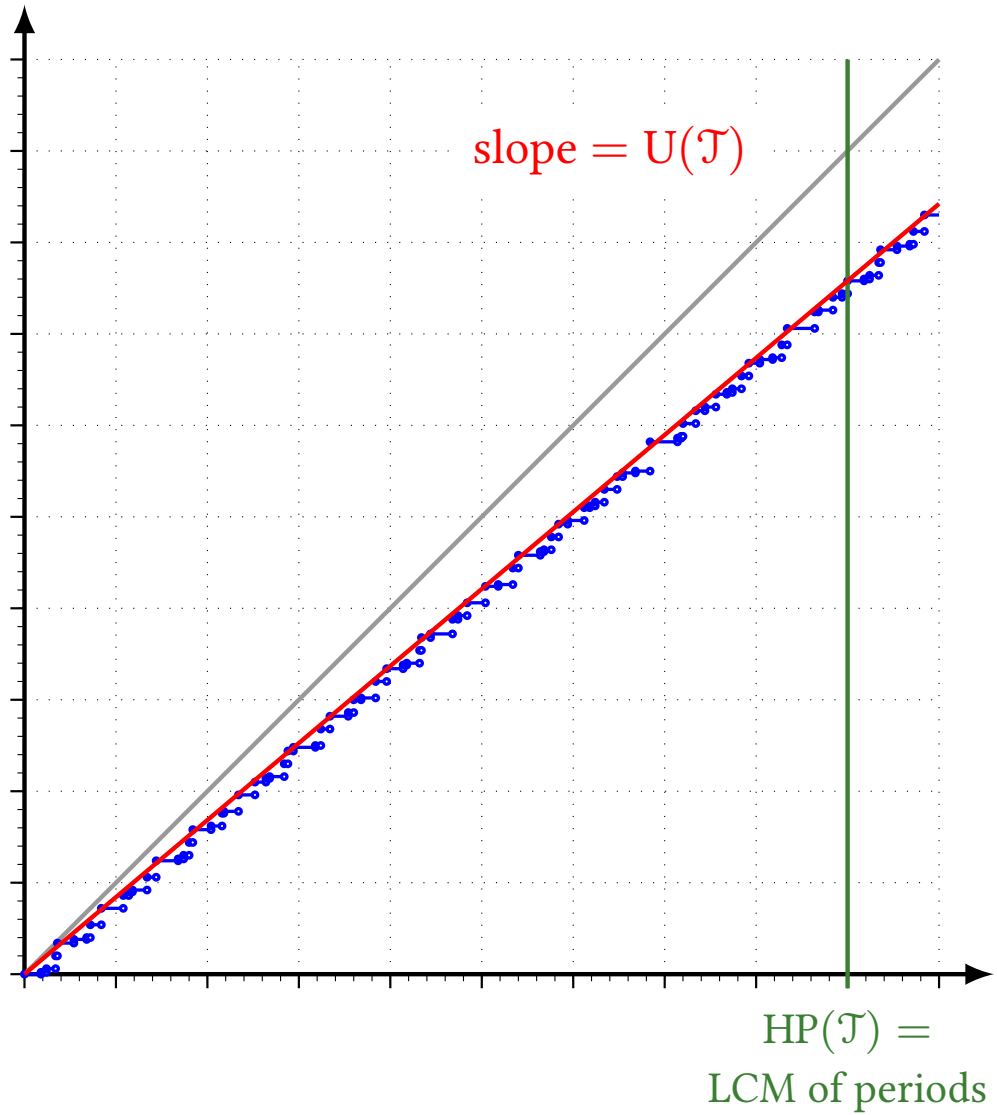
Recall the feasibility test for sporadic tasks



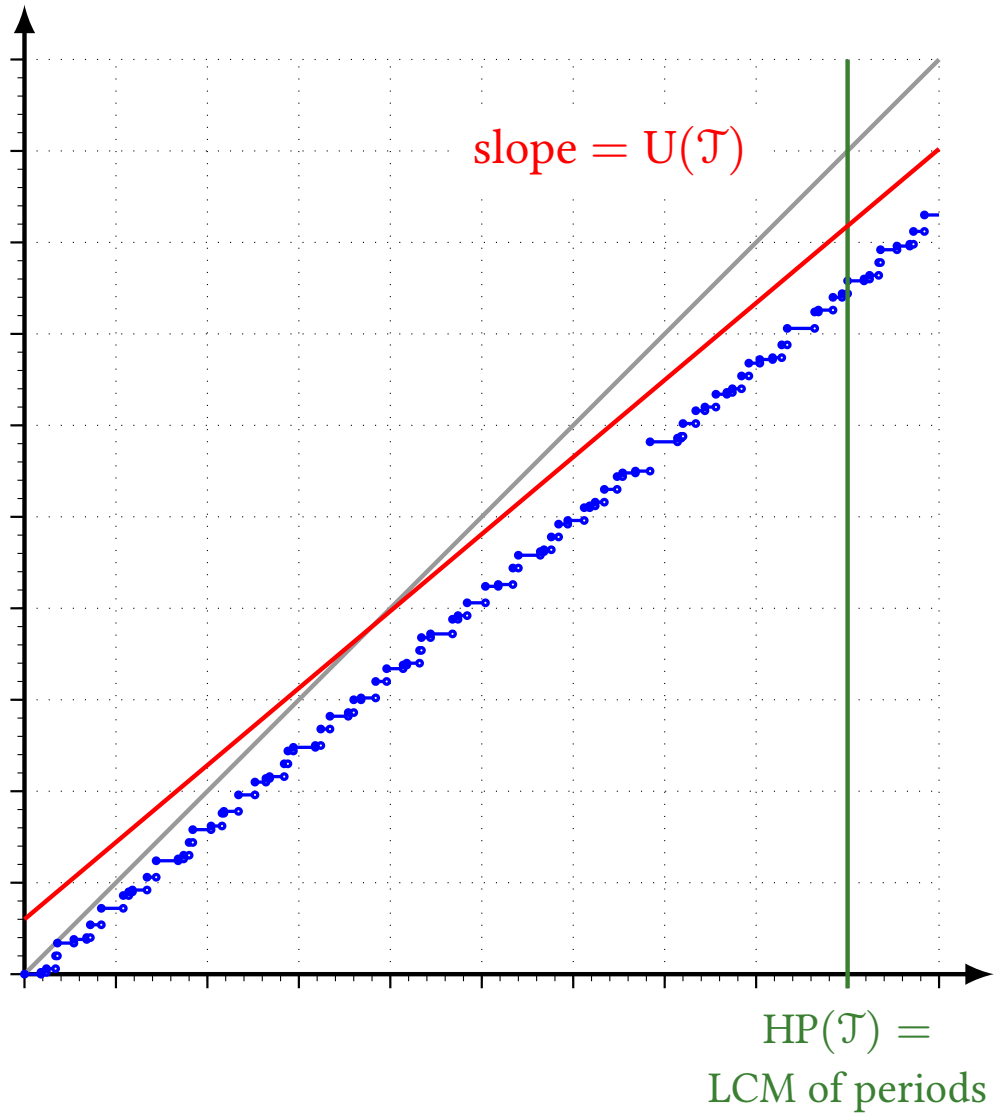
Recall the feasibility test for sporadic tasks



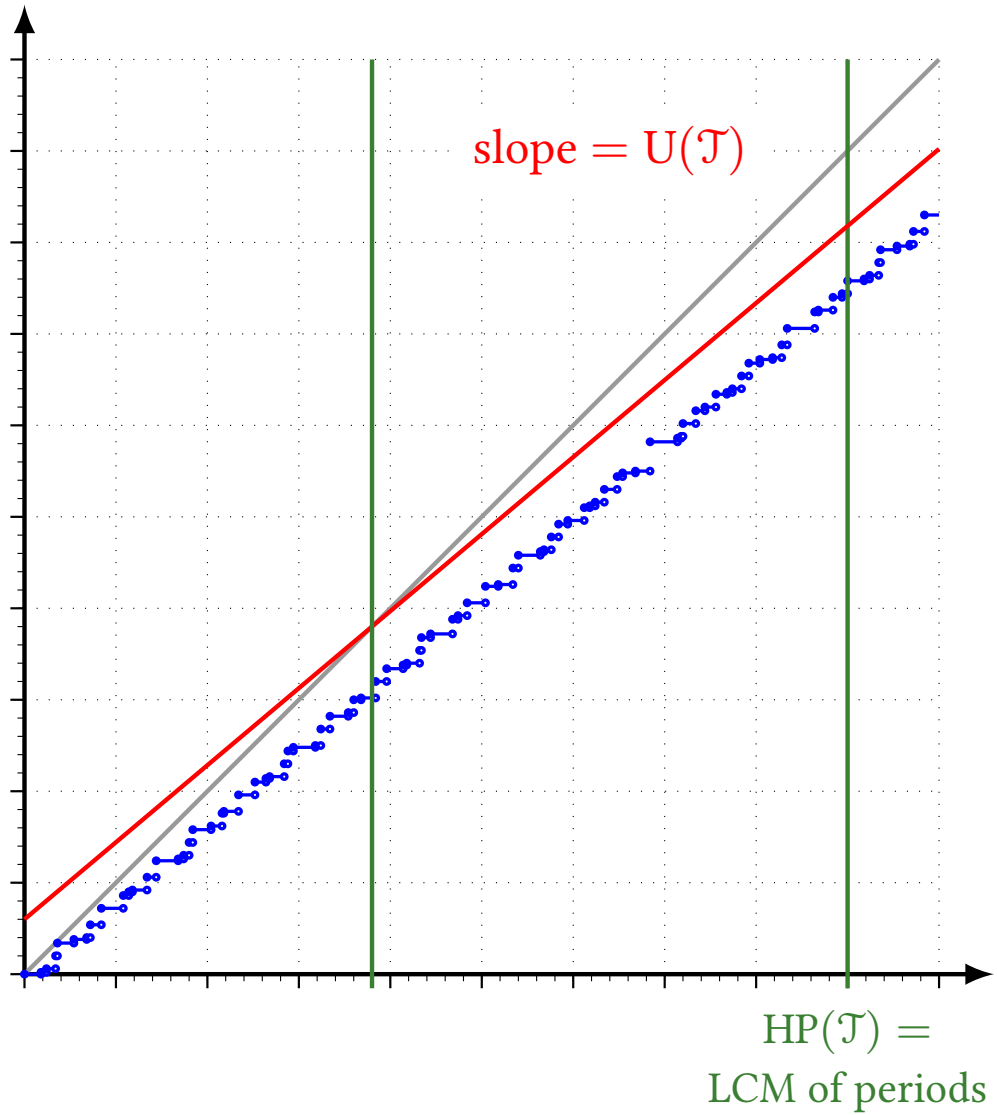
Recall the feasibility test for sporadic tasks



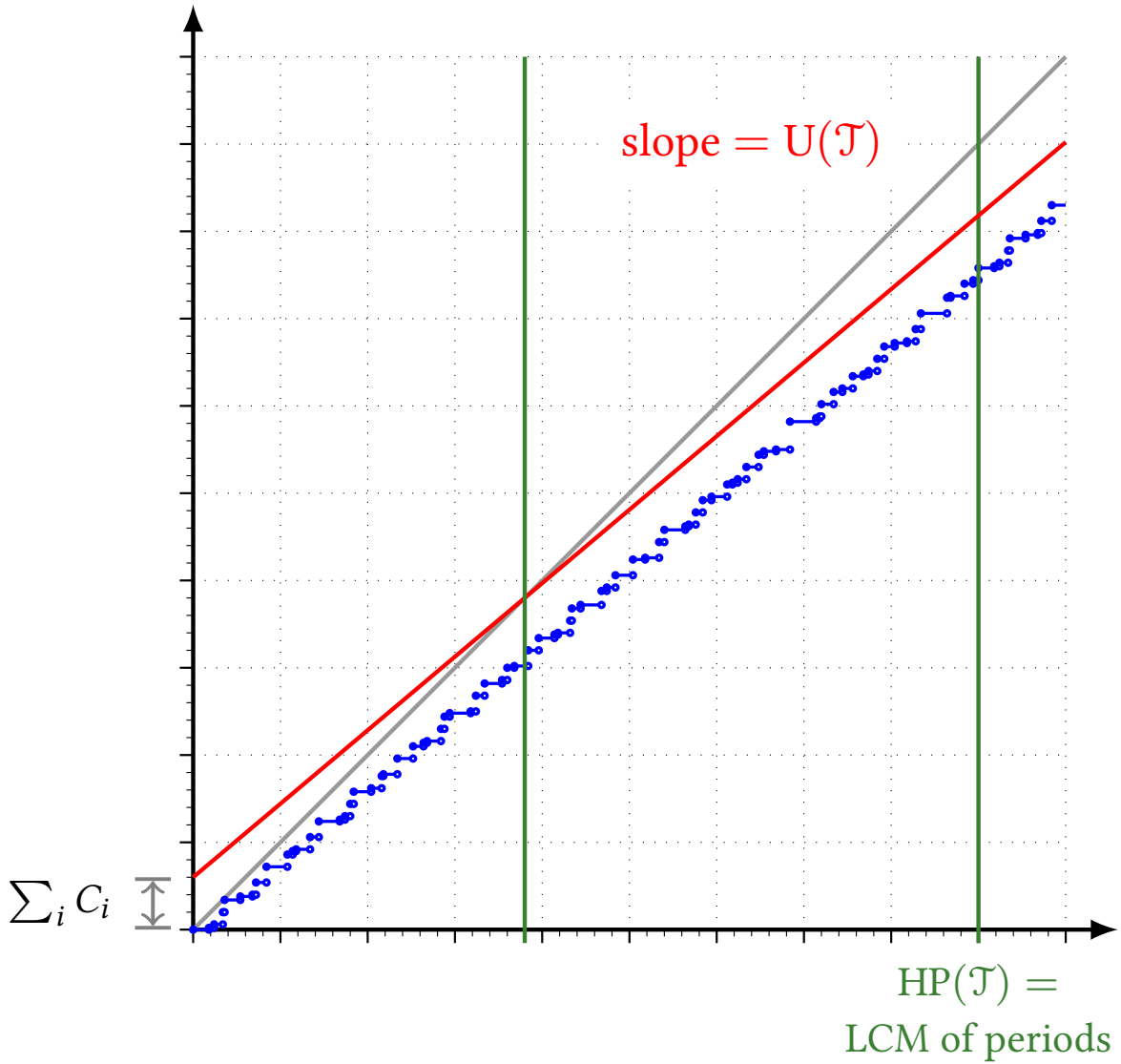
Recall the feasibility test for sporadic tasks



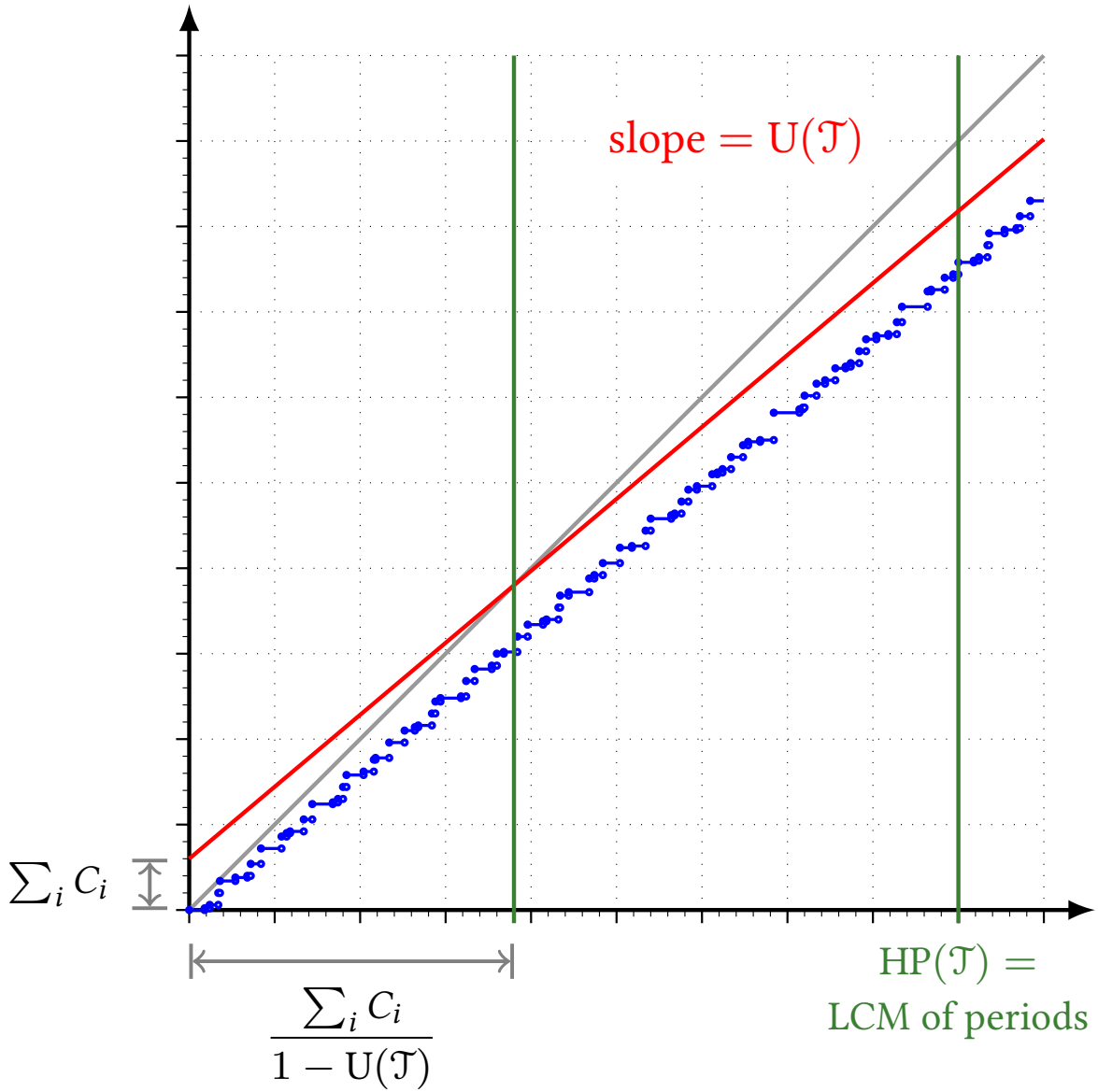
Recall the feasibility test for sporadic tasks



Recall the feasibility test for sporadic tasks



Recall the feasibility test for sporadic tasks



Generalizing sporadic tasks

- Example code structure for a periodic/sporadic task:

```
loop

    // Execute some function for, e.g.,
    // up to 11ms
    // (obtained via WCET analysis)

    delay until Previous_Period + 50ms;

end loop;
```

Generalizing sporadic tasks

- Example code structure for a periodic/sporadic task:

```
loop

    // Execute some function for, e.g.,
    // up to 11ms
    // (obtained via WCET analysis)

    delay until Previous_Period + 50ms;

end loop;
```

- What if the structure is more complicated?

A more complicated control structure

- Code is not always periodic:

```
loop

    // Execute some function
    delay until Period_Start + 50ms;
    // Execute another function
    delay until Prev_Function + 30ms;
    // Execute yet another function
    delay until Prev_Function + 70ms;

end loop;
```

A more complicated control structure

- Code is not always periodic:

```
loop

    // Execute some function
    delay until Period_Start + 50ms;
    // Execute another function
    delay until Prev_Function + 30ms;
    // Execute yet another function
    delay until Prev_Function + 70ms;

end loop;
```

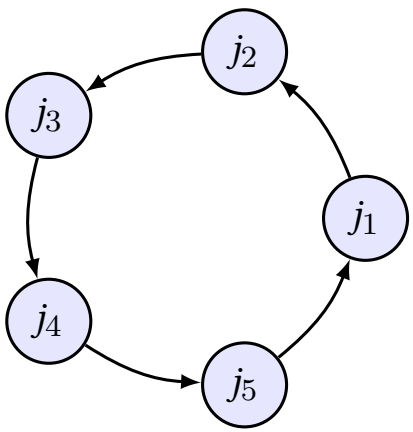
- Here a task is split in *frames*, each with own
  - Execution time
  - Inter-release separation until next frame
  - Relative deadline

The Generalized Multiframe (GMF) Task Model

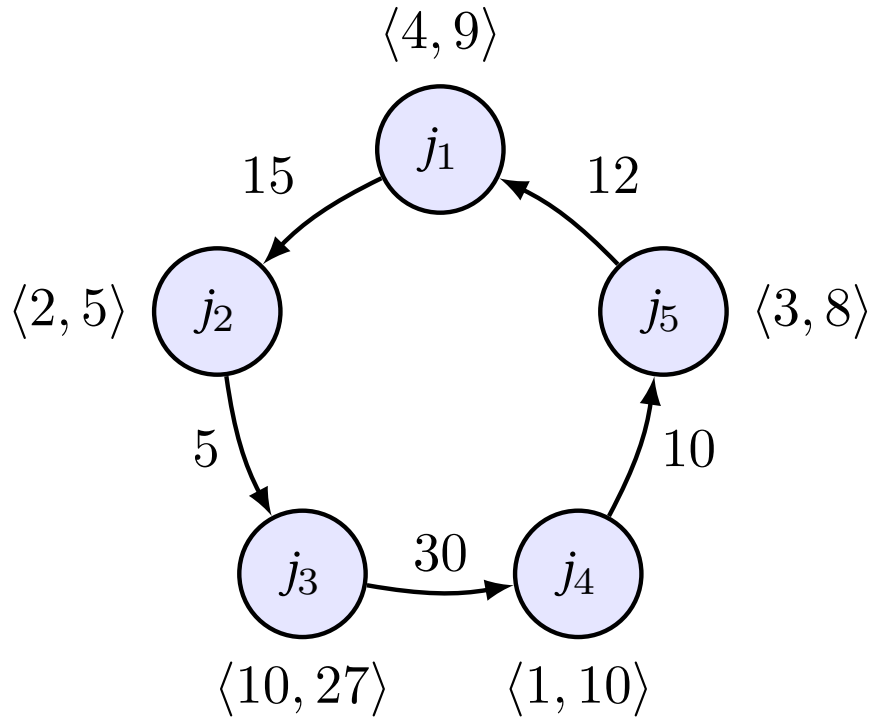
- Each task *cycles* through job types
  - Vector for WCET $(e^{(1)}, \ldots, e^{(n)})$
  - Vector for deadlines $(d^{(1)}, \ldots, d^{(n)})$
  - Vector for minimum inter-release delays $(p^{(1)}, \ldots, p^{(n)})$

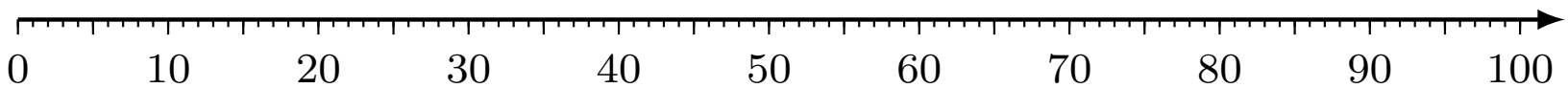
The Generalized Multiframe (GMF) Task Model

- Each task *cycles* through job types
  - Vector for WCET $(e^{(1)}, \ldots, e^{(n)})$
  - Vector for deadlines $(d^{(1)}, \ldots, d^{(n)})$
  - Vector for minimum inter-release delays $(p^{(1)}, \ldots, p^{(n)})$

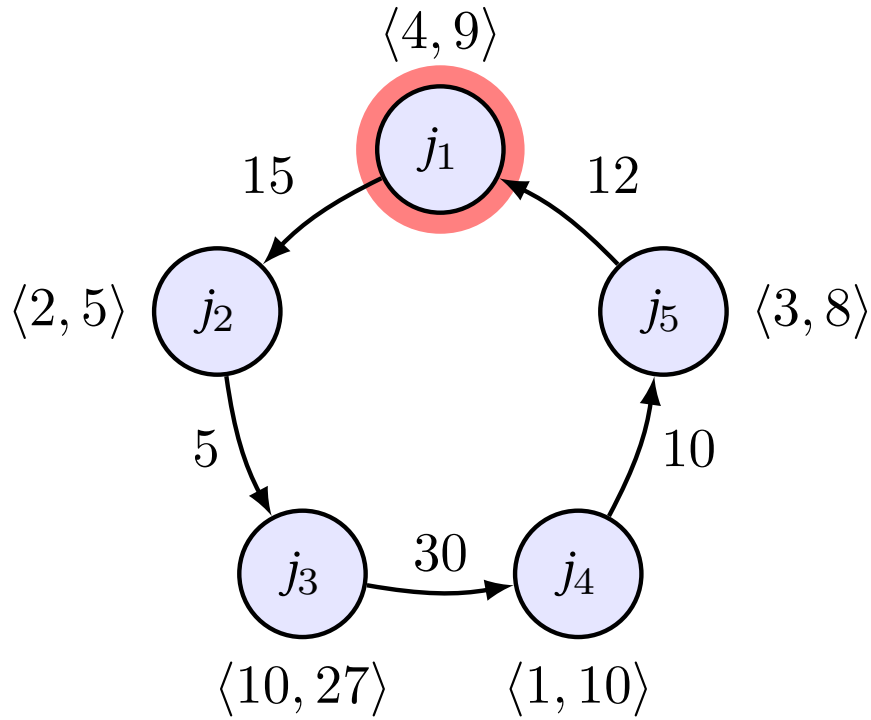


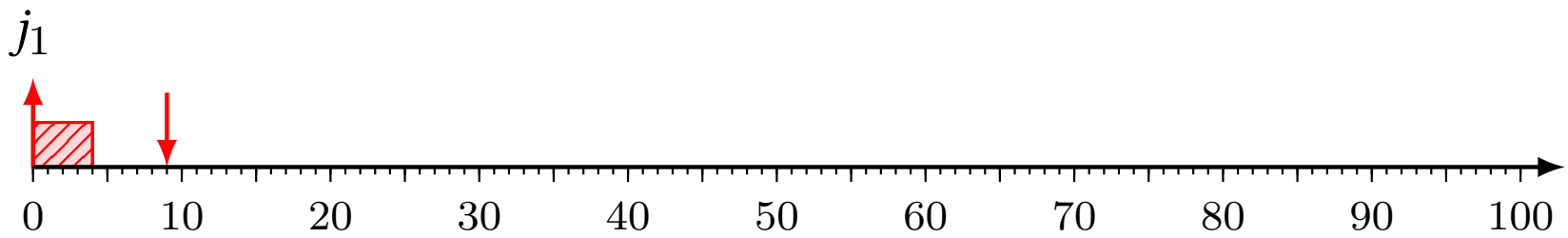
The Generalized Multiframe (GMF) Task Model



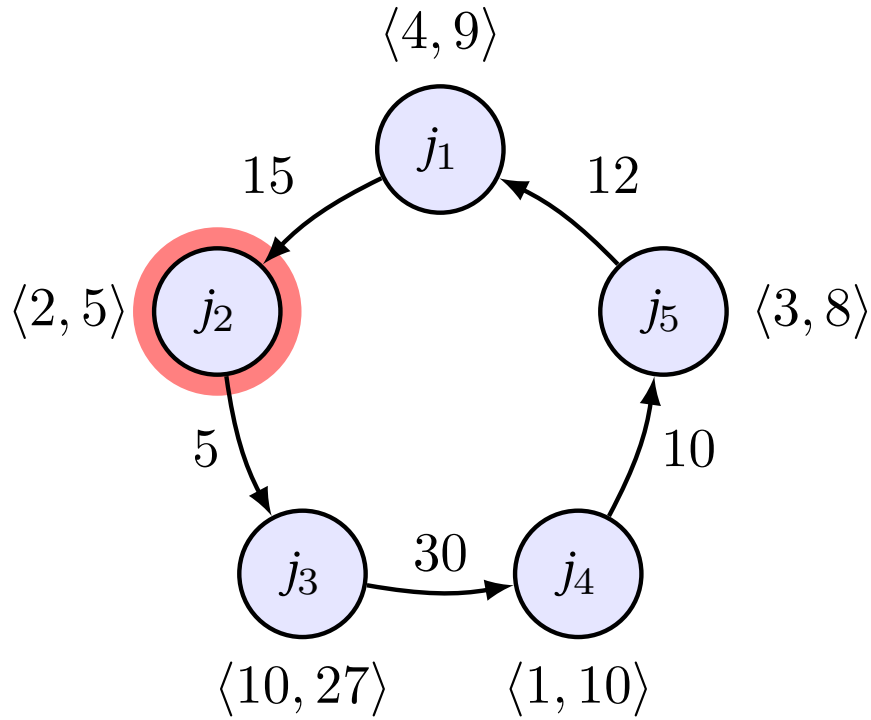


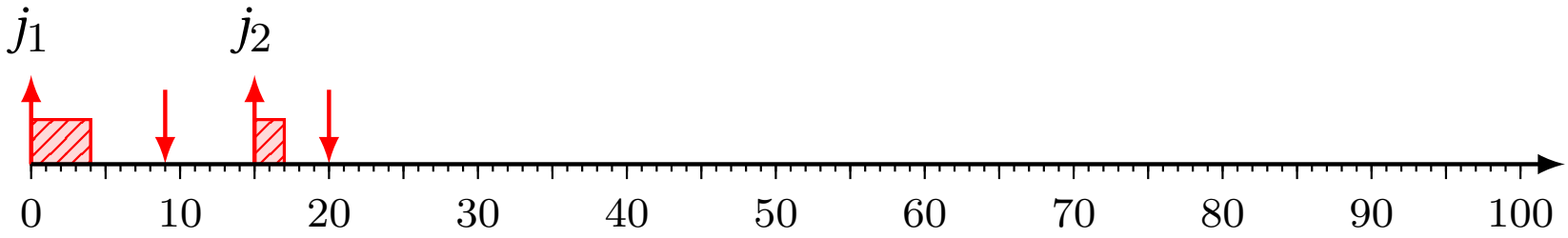
The Generalized Multiframe (GMF) Task Model



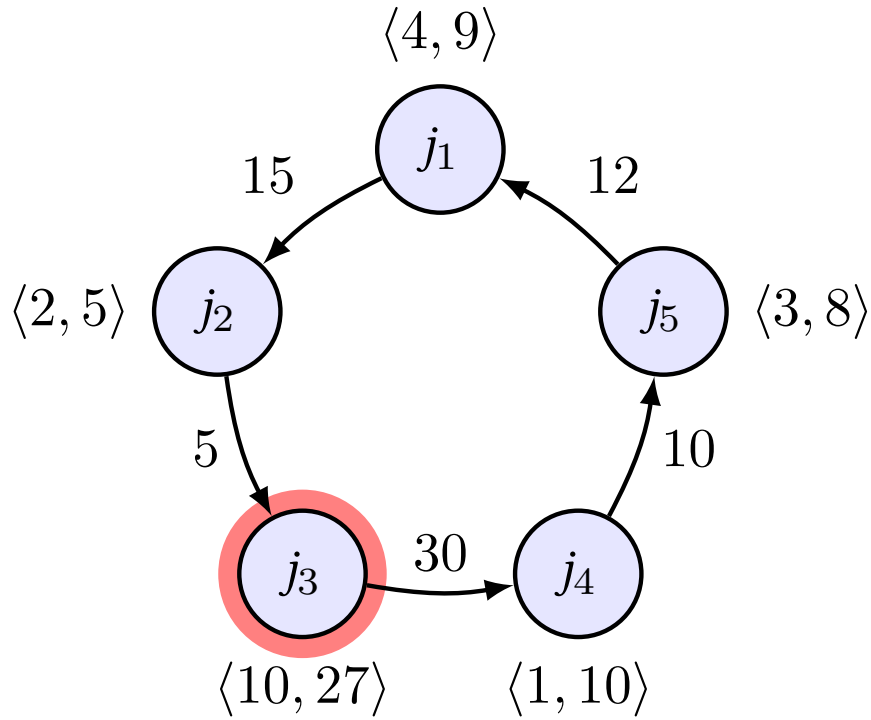


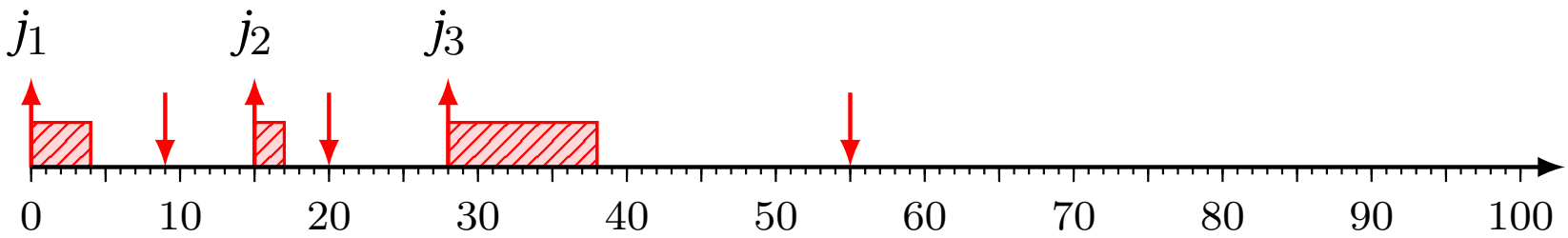
The Generalized Multiframe (GMF) Task Model



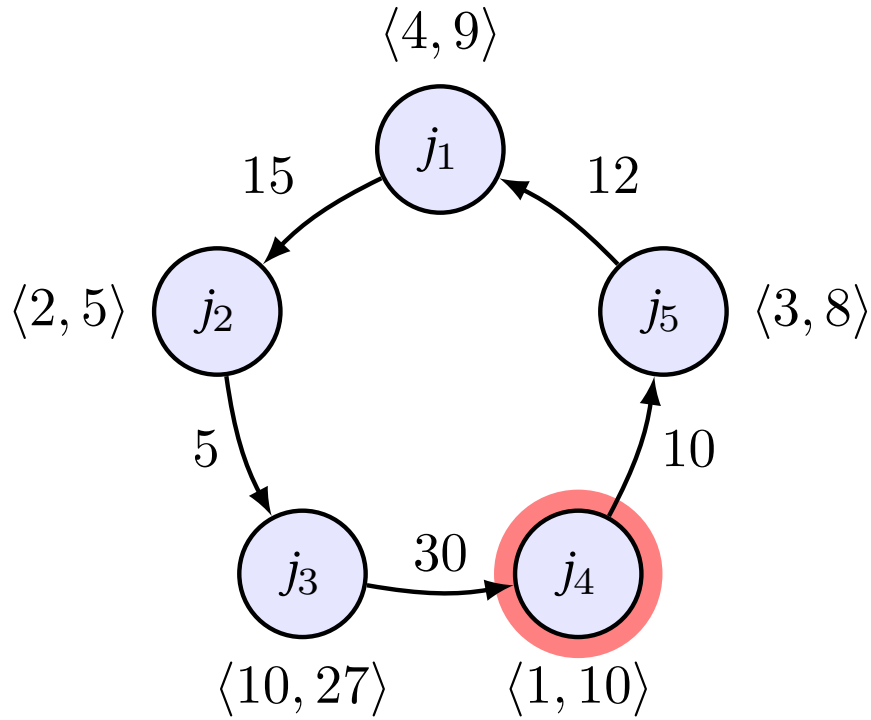


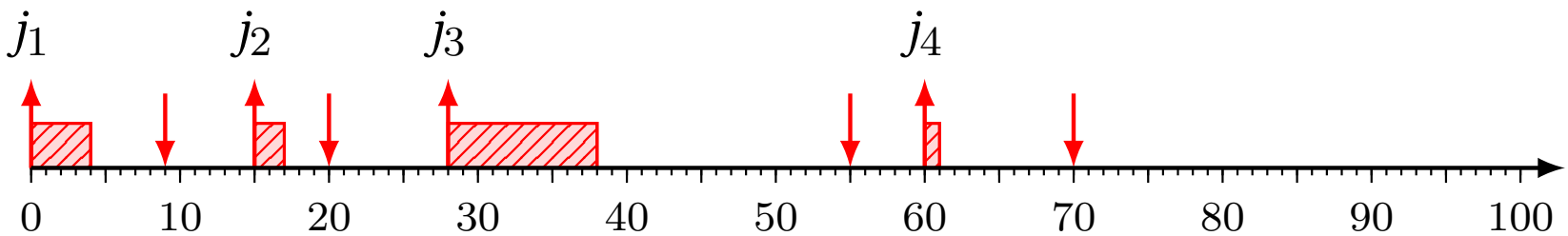
The Generalized Multiframe (GMF) Task Model



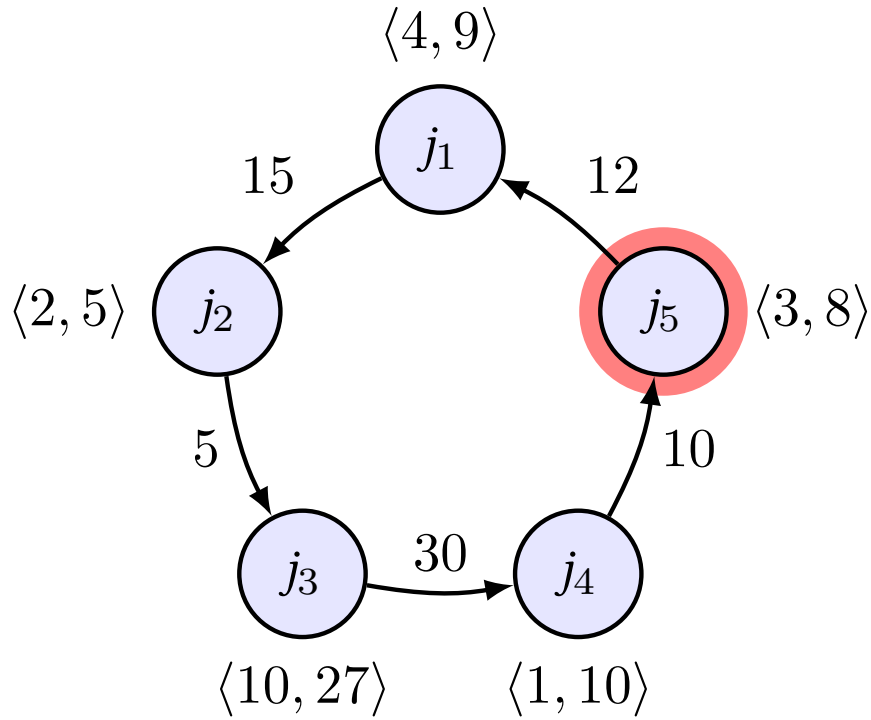


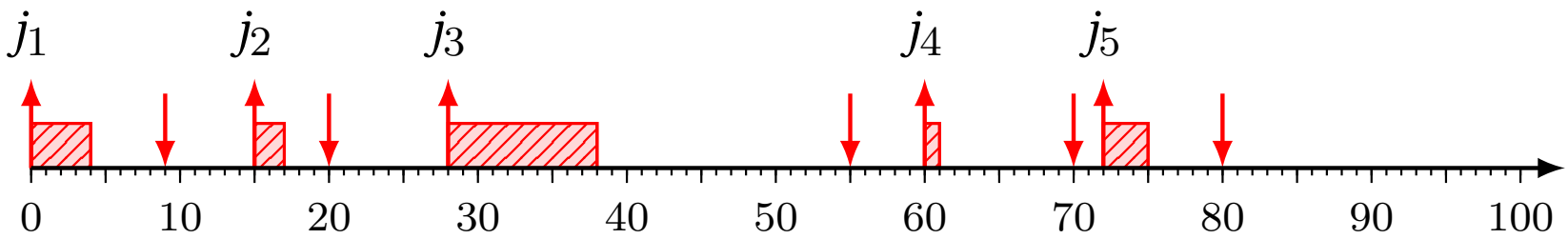
The Generalized Multiframe (GMF) Task Model



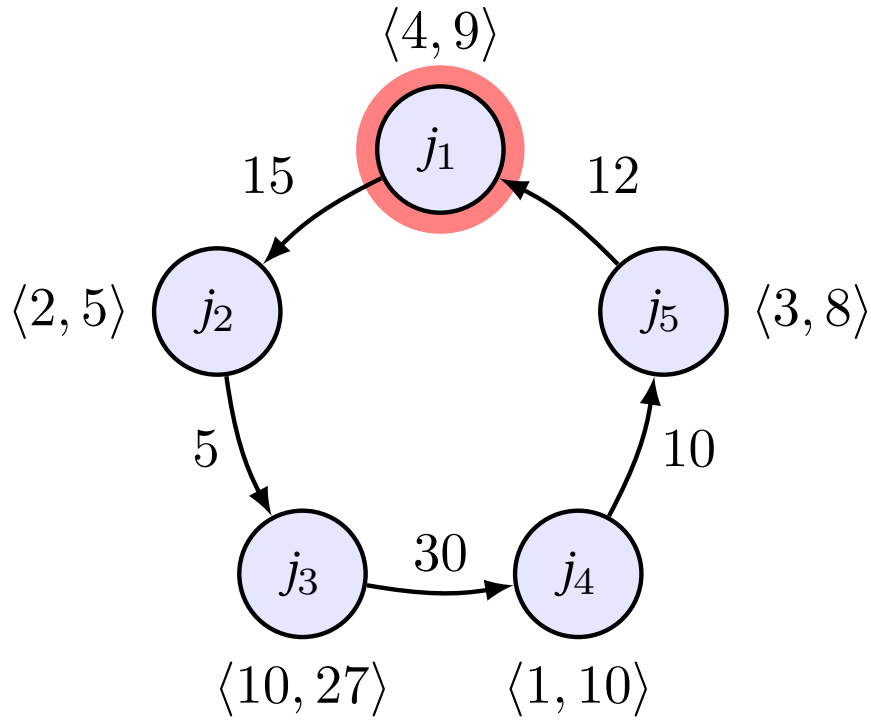


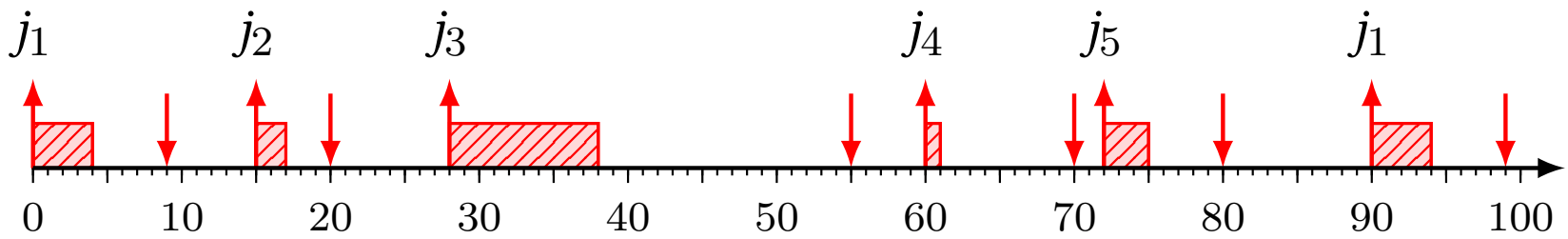
The Generalized Multiframe (GMF) Task Model





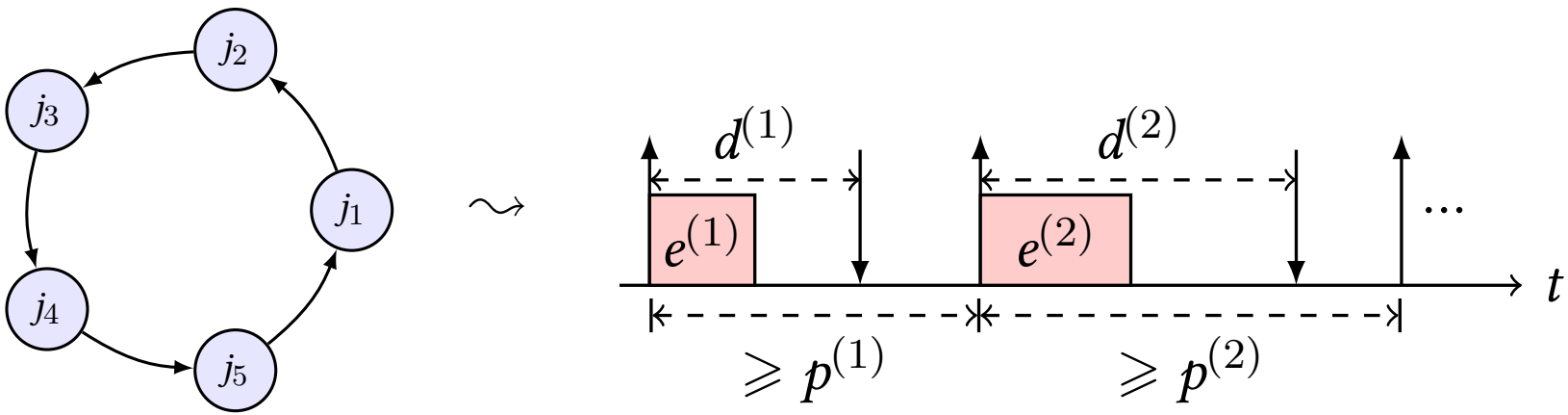
The Generalized Multiframe (GMF) Task Model





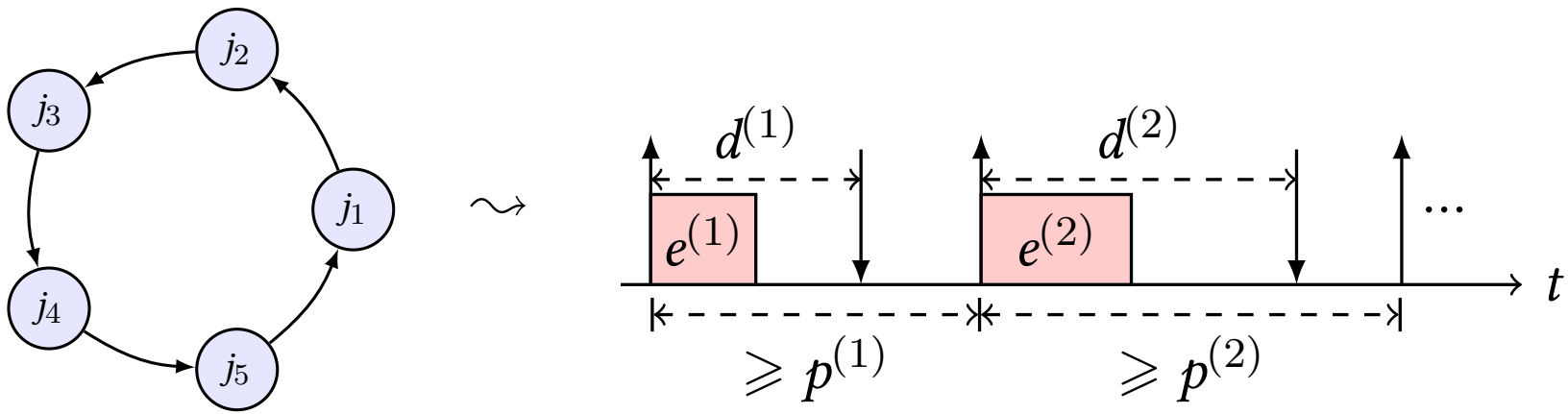
The Generalized Multiframe (GMF) Task Model

- Each task *cycles* through job types
  - Vector for WCET $(e^{(1)}, \ldots, e^{(n)})$
  - Vector for deadlines $(d^{(1)}, \ldots, d^{(n)})$
  - Vector for minimum inter-release delays $(p^{(1)}, \ldots, p^{(n)})$



The Generalized Multiframe (GMF) Task Model

- Each task *cycles* through job types
  - Vector for WCET $(e^{(1)}, \ldots, e^{(n)})$
  - Vector for deadlines $(d^{(1)}, \ldots, d^{(n)})$
  - Vector for minimum inter-release delays $(p^{(1)}, \ldots, p^{(n)})$



- Schedulability analysis for EDF?
  - Use the demand bound function, like for sporadic tasks
  - How to calculate dbf? What about the bound?
  - Exercise for the interested!
  - Read more in *Generalized Multiframe Tasks* (Baruah et al., 1999)

Generalize differently?

- What about *branches*?

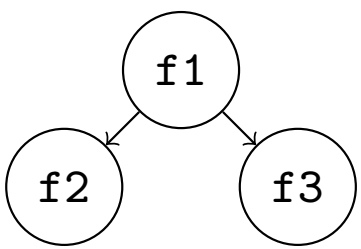
```
loop
    // Execute some function
    delay until Period_Start + 50ms;
    if (condition) then {
        // Execute another function
        delay until Prev_Function + 30ms;
    } else {
        // Execute yet another function
        delay until Prev_Function + 70ms;
    }
end loop;
```

Generalize differently?

- What about *branches*?

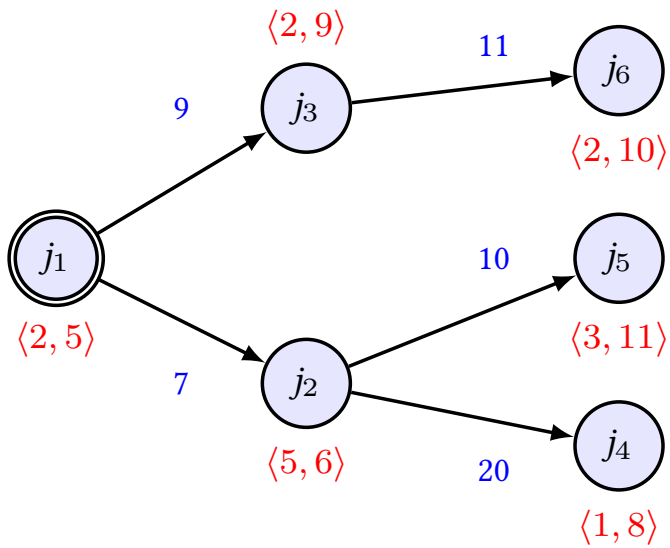
```
loop
    // Execute some function
    delay until Period_Start + 50ms;
    if (condition) then {
        // Execute another function
        delay until Prev_Function + 30ms;
    } else {
        // Execute yet another function
        delay until Prev_Function + 70ms;
    }
end loop;
```

- Each task is a *tree*
  - Vertices represent jobs
  - Edges represent control flow and delays
  - Restarted once a leaf is reached



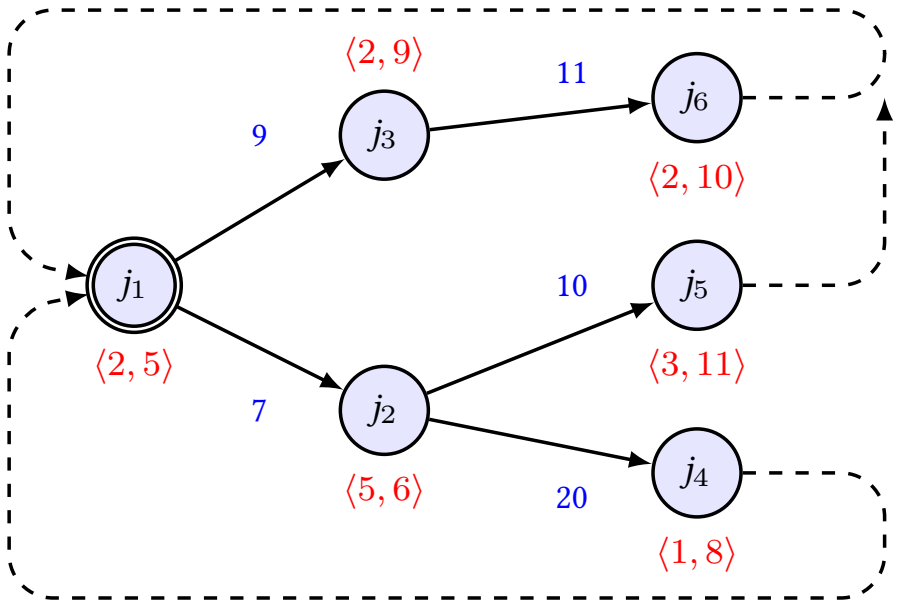
The Recurring Branching (RB) Task Model

- Introduces *branching* structures
- A *tree* for each task
  - Vertices j: job types with WCET and deadline $\langle e(j), d(j) \rangle$
  - Edges (j_i, j_k): minimum inter-release delays $p(j_i, j_k)$



The Recurring Branching (RB) Task Model

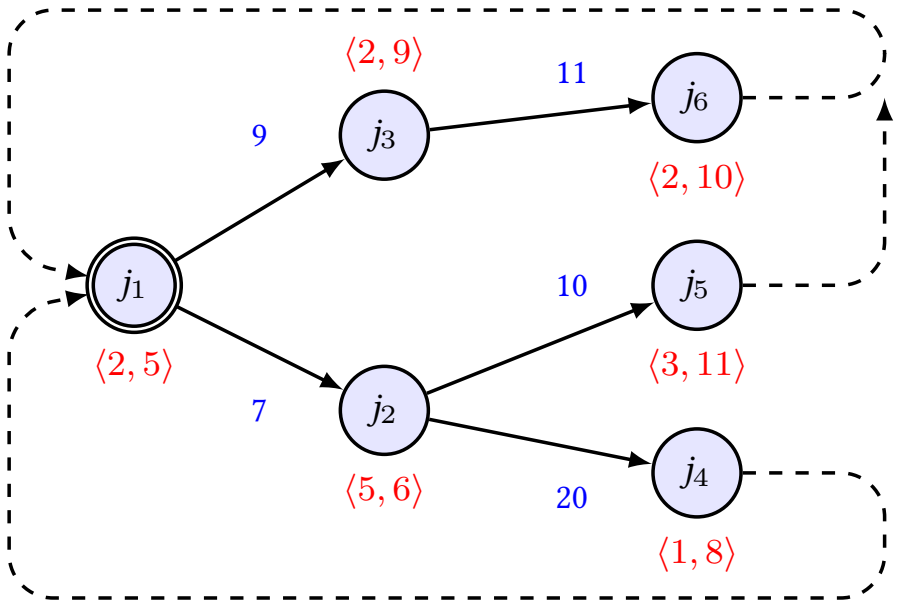
- Introduces *branching* structures
- A *tree* for each task
  - Vertices j: job types with WCET and deadline $\langle e(j), d(j) \rangle$
  - Edges (j_i, j_k): minimum inter-release delays $p(j_i, j_k)$
  - General period parameter P



Period $P = 57$

The Recurring Branching (RB) Task Model

- Introduces *branching* structures
- A *tree* for each task
  - Vertices j: job types with WCET and deadline $\langle e(j), d(j) \rangle$
  - Edges (j_i, j_k): minimum inter-release delays $p(j_i, j_k)$
  - General period parameter P



- Feasibility analysis is also dbf-based, rather involved
- Read more in *Feasibility analysis of recurring branching tasks* (Baruah, 1998)

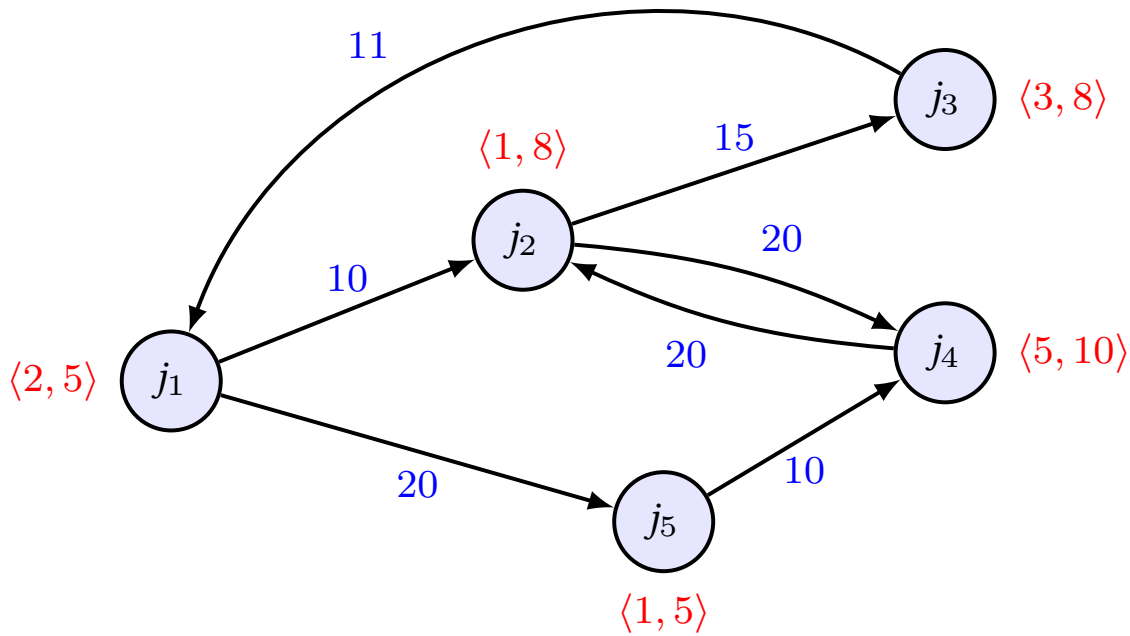
Generalize further?

- Now we can model:
  - Periodic behavior
  - Multiple frames
  - Branching behavior
- Still not possible:
  - Local loops
  - Local modes
  - ...

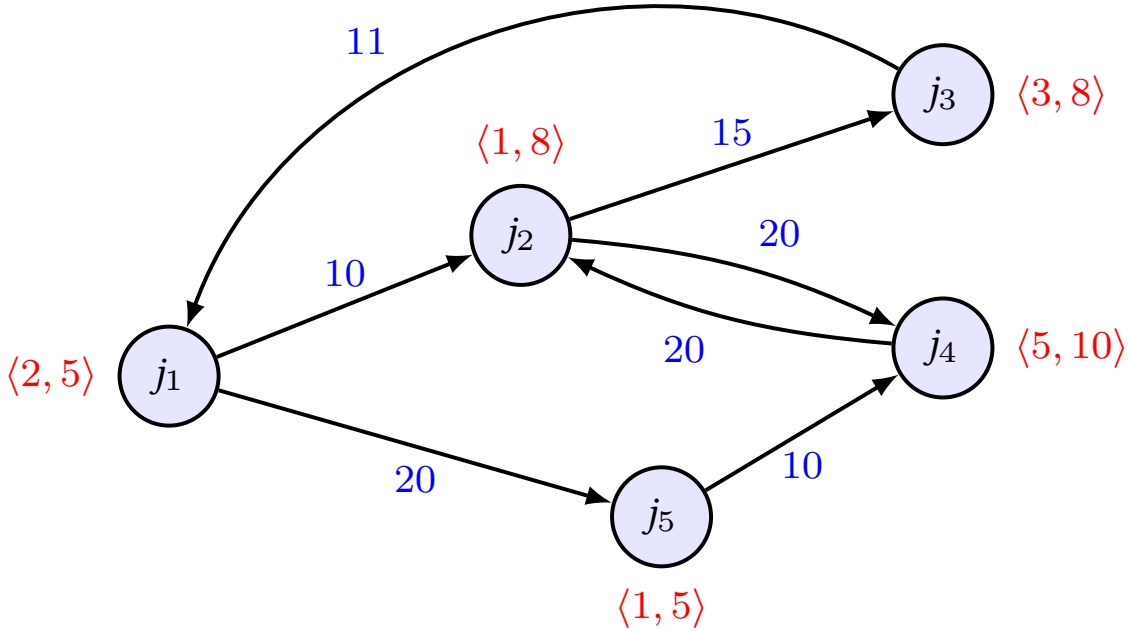
```
loop
    f1();
    delay ...
    if (cond) then {
        while (cond2) {
            f2();
            delay ...
        }
    } else {
        f3();
        delay ...
    }
end loop;
```

The Digraph Real-Time (DRT) Task Model

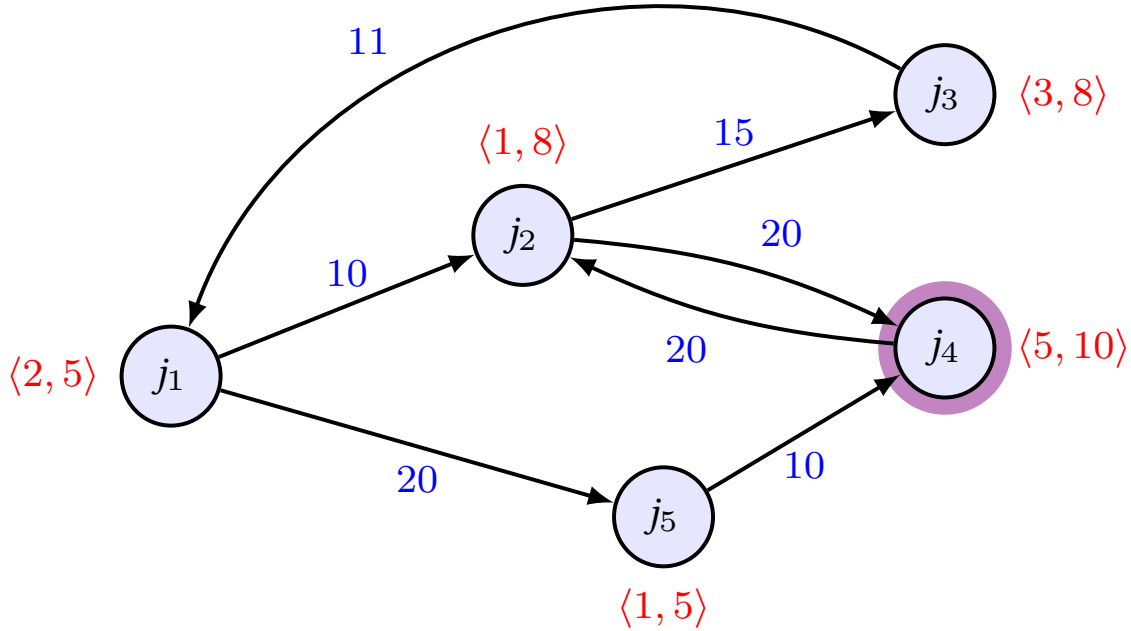
- Generalizes sporadic, GMF, RRT (almost), ...
- *Directed graph* for each task
  - Vertices j: job types with WCET and deadline $\langle e(j), d(j) \rangle$
  - Edges (j_i, j_k): minimum inter-release delays $p(j_i, j_k)$



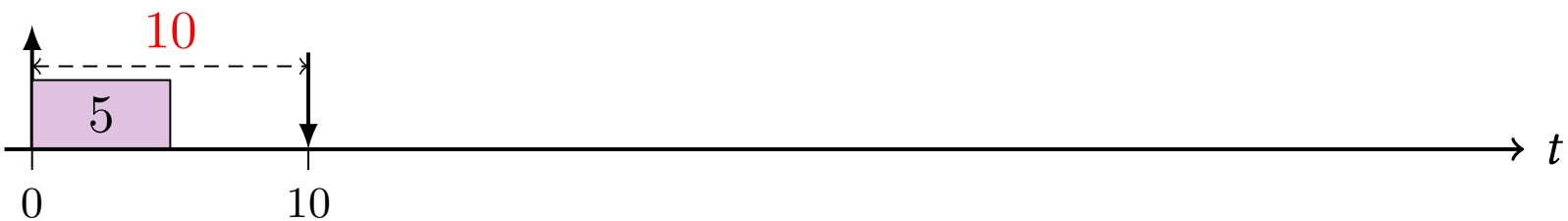
DRT: Semantics



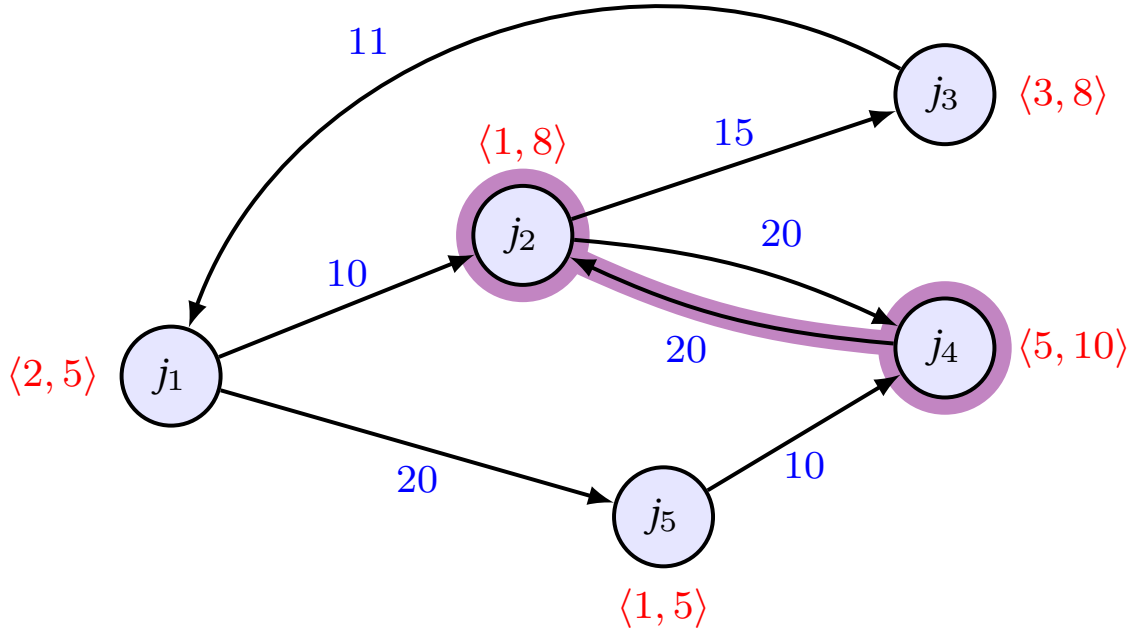
DRT: Semantics



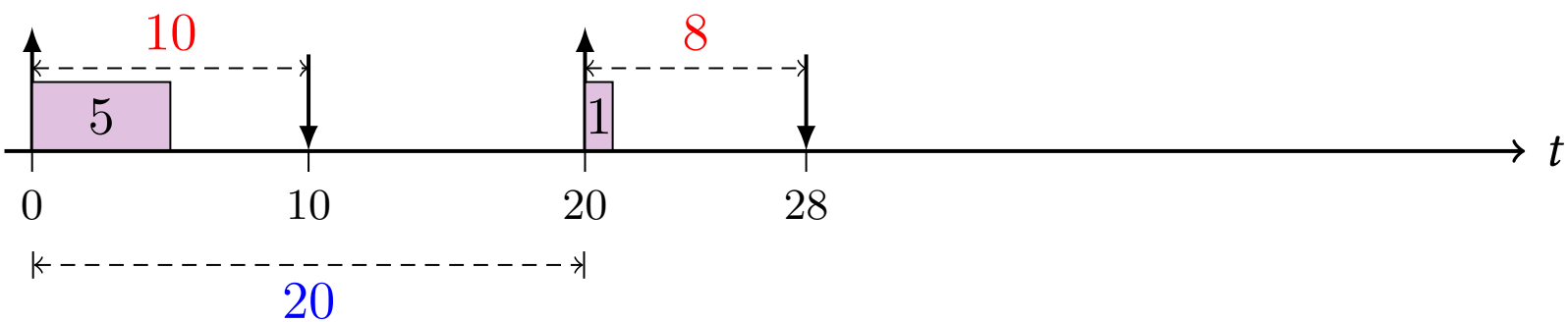
Path $\pi = (j_4)$



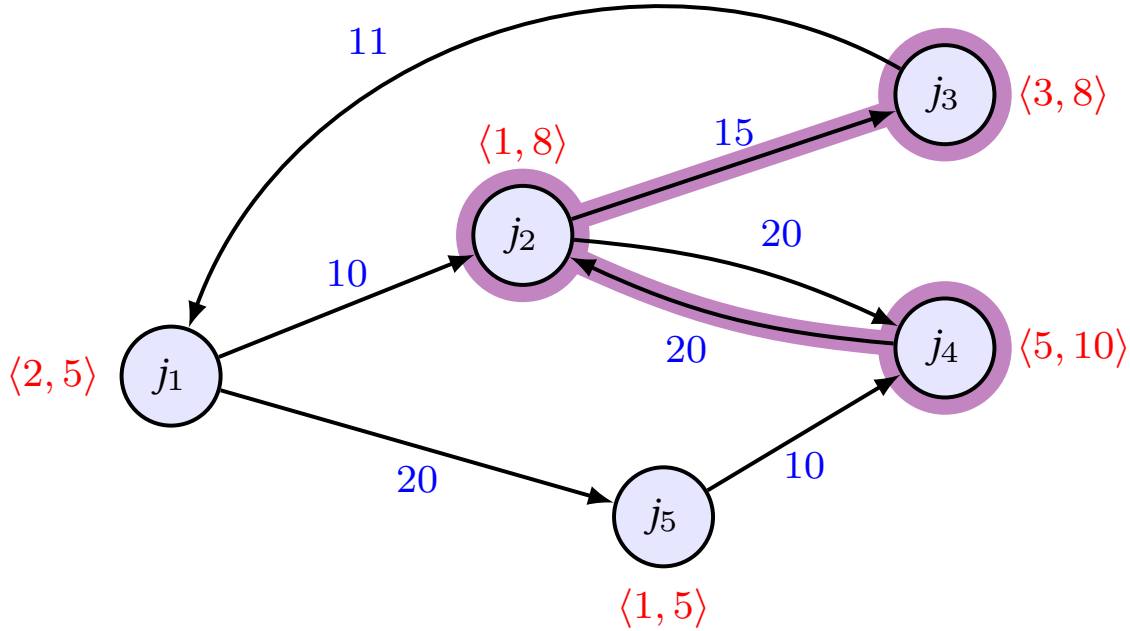
DRT: Semantics



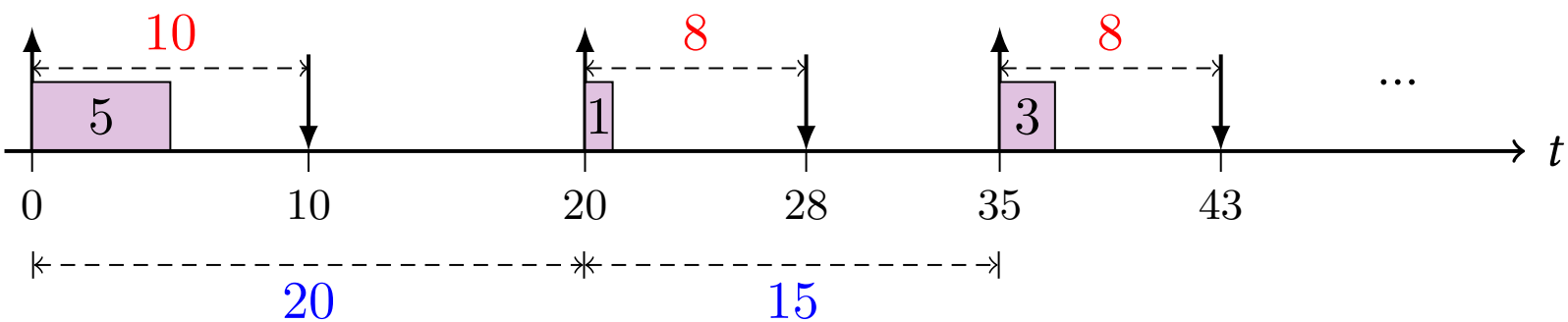
Path $\pi = (j_4, j_2)$



DRT: Semantics



Path $\pi = (j_4, j_2, j_3)$



DRT: Feasibility analysis

- The main result about demand bound functions still holds:

Theorem

A task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of DRT tasks is feasible on a single preemptive processor iff

$$\forall t \geqslant 0 : \qquad \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}(\tau_i, t) \leqslant t.$$

DRT: Feasibility analysis

- The main result about demand bound functions still holds:

Theorem

A task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of DRT tasks is feasible on a single preemptive processor iff

$$\forall t \geqslant 0 : \qquad \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}(\tau_i, t) \leqslant t.$$

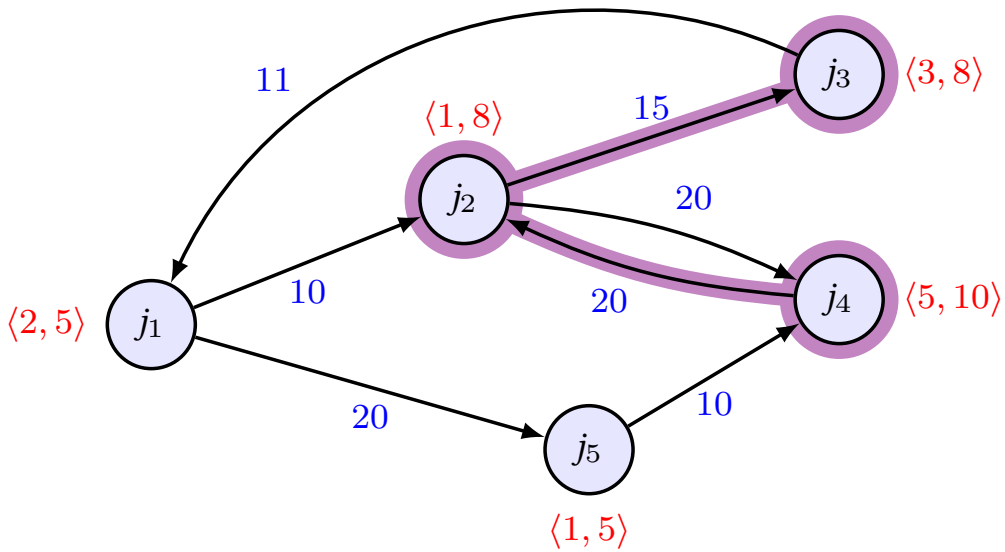
- Thus, do as before:
  1. Compute demand bound functions $\mathrm{dbf}(\tau_i, t)$.
     - How to do that for a given t?
  2. Test the inequality for all $t \leqslant B$ for some bound B.
     - How to derive the bound B?

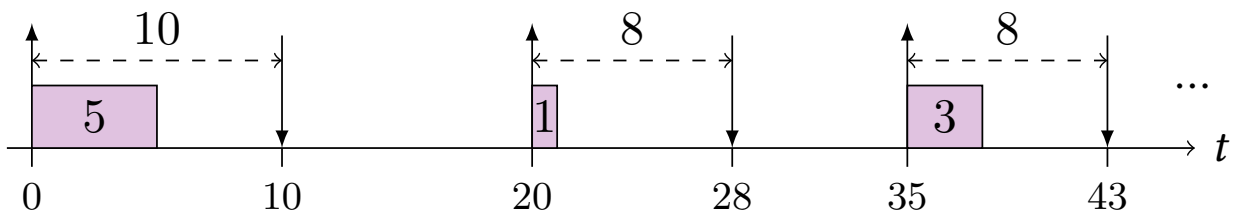
Demand pairs

- Recall demand bound function:
  - Interval length t, sum all demand …
  - … of jobs *released* and with *deadline* inside

Demand pairs

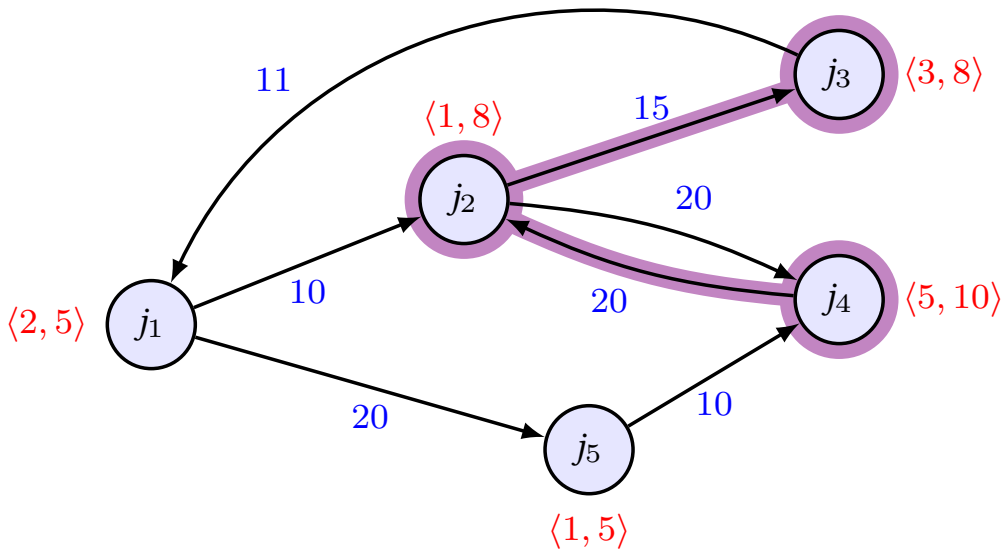
- Recall demand bound function:
  - Interval length t, sum all demand …
  - … of jobs *released* and with *deadline* inside
- Consider a path in a task's graph:





Demand pairs

- Recall demand bound function:
  - Interval length t, sum all demand …
  - … of jobs *released* and with *deadline* inside
- Consider a path in a task's graph:



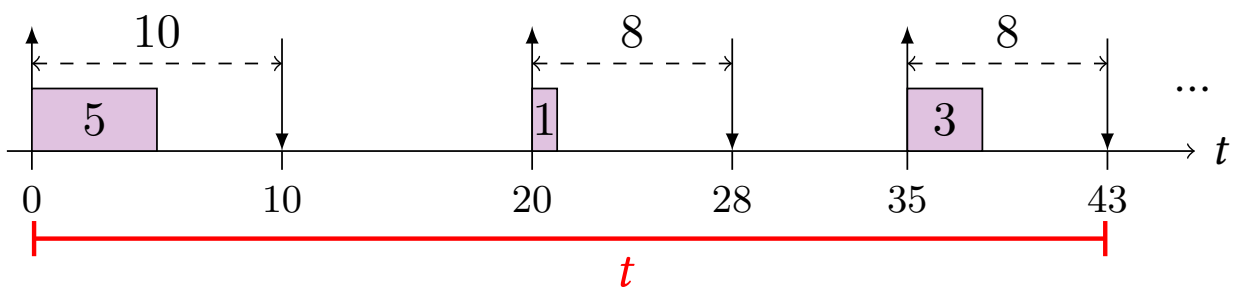
Execution demand:

$5 + 1 + 3 = 9$

Interval size:

$20 + 15 + 8 = 43$

Demand pair $\langle 9, 43\rangle$

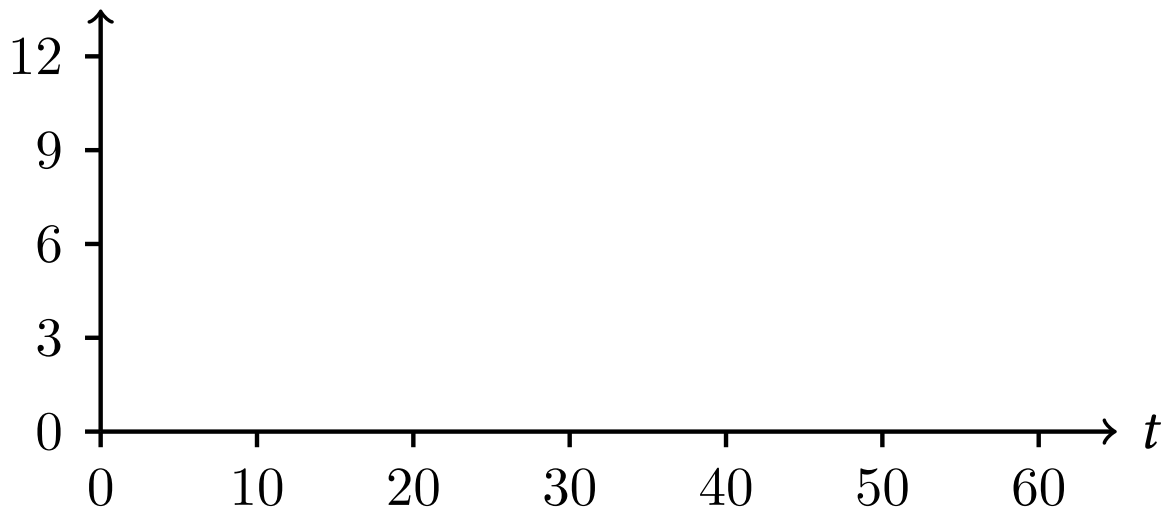


Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand …
  - … during interval of size d

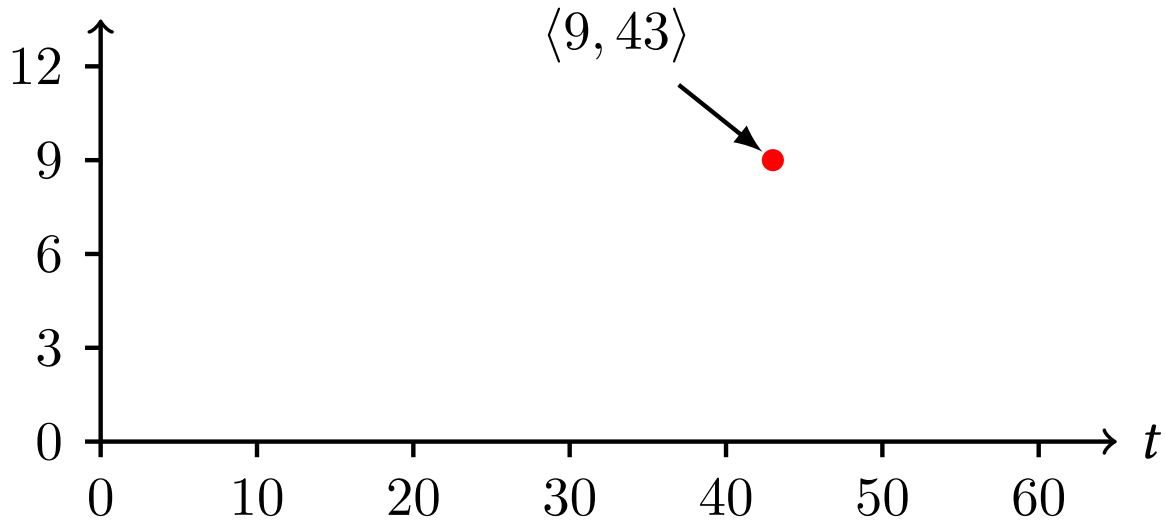
Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand ...
  - ... during interval of size d
- Useful for the demand bound function!



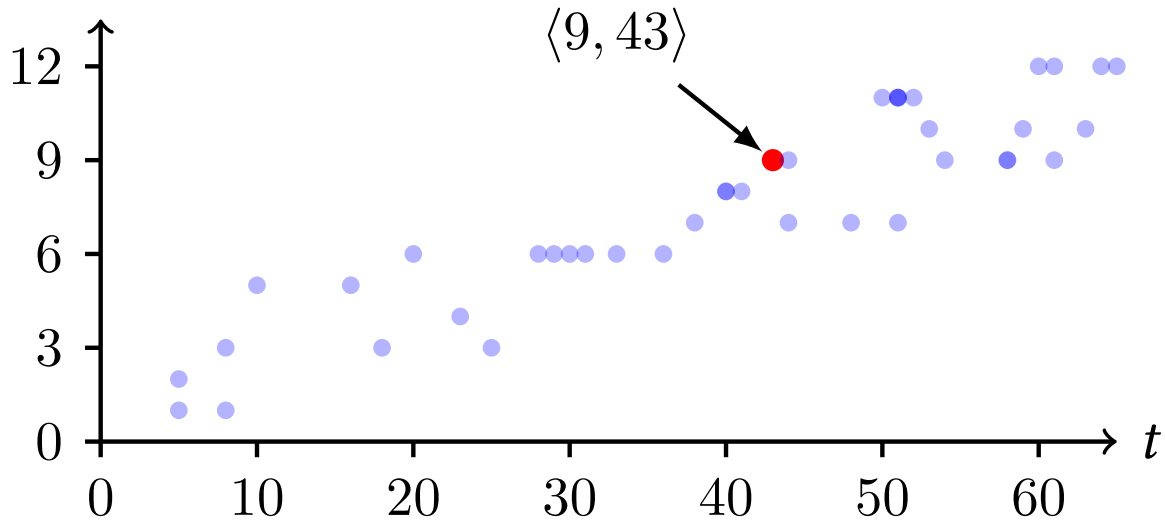
Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand ...
  - ... during interval of size d
- Useful for the demand bound function!



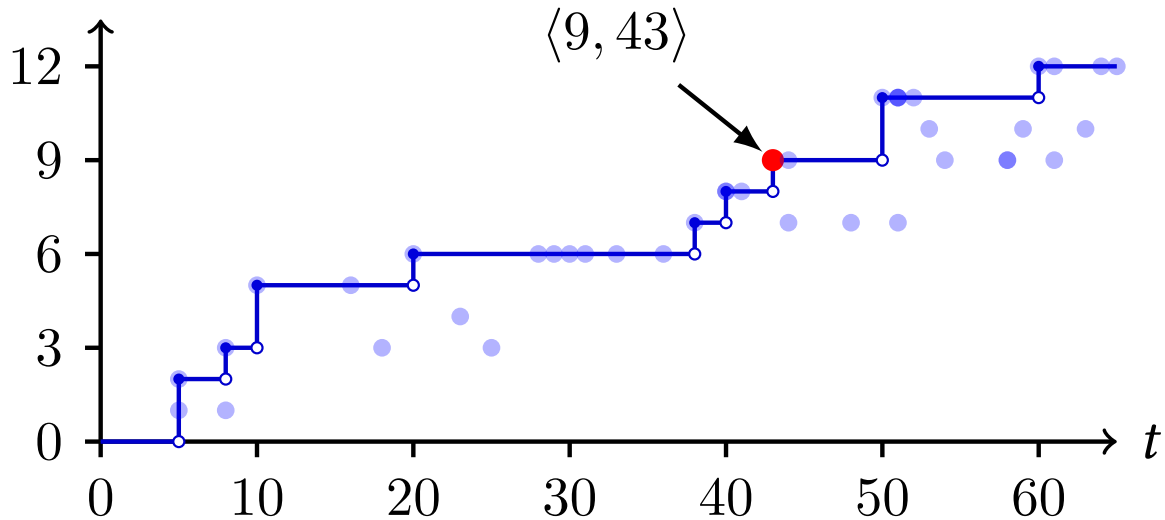
Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand ...
  - ... during interval of size d
- Useful for the demand bound function!



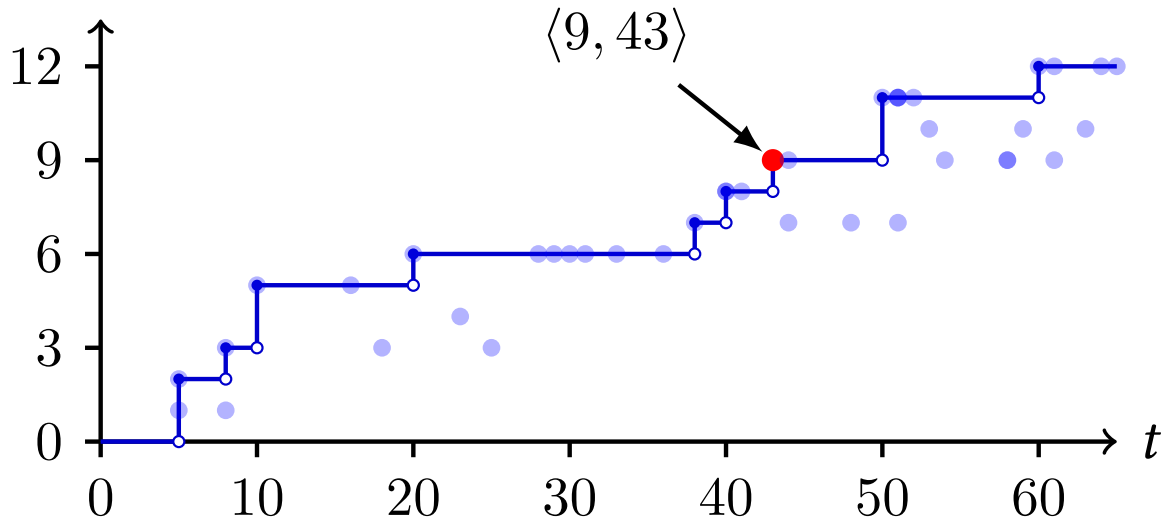
Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand ...
  - ... during interval of size d
- Useful for the demand bound function!



Demand pairs (cont.)

- From demand pair $\langle e, d \rangle$ we learn:
  - Task can create e units of exec. demand …
  - … during interval of size d
- Useful for the demand bound function!



- Thus: Compute *all demand pairs*, then take "maximum"

$$\mathrm{dbf}(\tau_i, t) = \max \{ e \mid \langle e, d \rangle \text{ demand pair with } d \leqslant t \}$$

Demand pairs (cont. 2)

More formally

- Given path $\pi = (\pi_1, \ldots, \pi_k)$
- *Execution demand:* $e(\pi) := \sum_{i=1}^{k} e(\pi_i)$
- *Deadline:* $d(\pi) := \sum_{i=1}^{k-1} p(\pi_i, \pi_{i+1}) + d(\pi_k)$
- $\langle e(\pi), d(\pi) \rangle$ is a *demand pair* for π

Demand pairs (cont. 2)

More formally

- Given path $\pi = (\pi_1, \ldots, \pi_k)$
- *Execution demand:* $e(\pi) := \sum_{i=1}^{k} e(\pi_i)$
- *Deadline:* $d(\pi) := \sum_{i=1}^{k-1} p(\pi_i, \pi_{i+1}) + d(\pi_k)$
- $\langle e(\pi), d(\pi) \rangle$ is a *demand pair* for π

How to compute all demand pairs?

- Enumerate all paths?

Demand pairs (cont. 2)

More formally

- Given path $\pi = (\pi_1, \ldots, \pi_k)$
- *Execution demand:* $e(\pi) := \sum_{i=1}^{k} e(\pi_i)$
- *Deadline:* $d(\pi) := \sum_{i=1}^{k-1} p(\pi_i, \pi_{i+1}) + d(\pi_k)$
- $\langle e(\pi), d(\pi) \rangle$ is a *demand pair* for π

How to compute all demand pairs?

- Enumerate all paths? Too expensive! (Exponential)

Demand pairs (cont. 2)

More formally

- Given path $\pi = (\pi_1, \ldots, \pi_k)$
- *Execution demand:* $e(\pi) := \sum_{i=1}^{k} e(\pi_i)$
- *Deadline:* $d(\pi) := \sum_{i=1}^{k-1} p(\pi_i, \pi_{i+1}) + d(\pi_k)$
- $\langle e(\pi), d(\pi) \rangle$ is a *demand pair* for π

How to compute all demand pairs?

- Enumerate all paths? Too expensive! (Exponential)
- Better: Iteration using *abstraction*
- (Remark: Demand pairs are abstractions of paths)

Demand triples

- Idea: Start with 0-paths (one vertex), extend stepwise

Demand triples

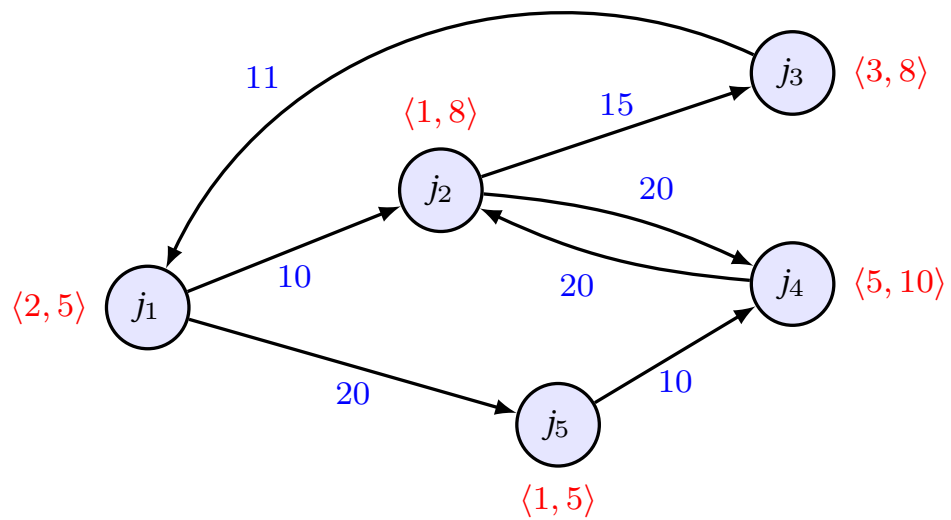
- Idea: Start with 0-paths (one vertex), extend stepwise
- We need: Abstraction which
  1. allows to *extend* paths,
  2. contains demand pair information,
  3. without visiting/storing all paths

Demand triples

- Idea: Start with 0-paths (one vertex), extend stepwise
- We need: Abstraction which
  1. allows to *extend* paths,
  2. contains demand pair information,
  3. without visiting/storing all paths
- Idea: *Demand triples*
  - Execution demand $e(\pi)$
  - Deadline $d(\pi)$
  - Last vertex π_k

Demand triples

- Idea: Start with 0-paths (one vertex), extend stepwise
- We need: Abstraction which
  1. allows to *extend* paths,
  2. contains demand pair information,
  3. without visiting/storing all paths
- Idea: *Demand triples*
  - Execution demand $e(\pi)$
  - Deadline $d(\pi)$
  - Last vertex π_k
- Demand triple $\langle e(\pi), d(\pi), \pi_k \rangle$ is another path abstraction!



Path (j_4)
$\rightsquigarrow \langle 5, 10, j_4 \rangle$

Path (j_4, j_2)
$\rightsquigarrow \langle 6, 28, j_2 \rangle$

Path (j_4, j_2, j_3)
$\rightsquigarrow \langle 9, 43, j_3 \rangle$

Iterative procedure

- Create all demand triples up to bound B:
  1. Store all *0-paths*, i.e., $\langle e(j), d(j), j \rangle$ for all vertices j
  2. Pick some stored unmarked demand triple $\langle e, d, j_i \rangle$, then *mark* it
  3. *Create new demand triples*:
     - For each successor vertex j_k of j_i
     - $e' = e + e(j_k)$
     - $d' = d - d(j_i) + p(j_i, j_k) + d(j_k)$
     - $\langle e', d', j_k \rangle$ is a new demand triple!
  4. Store each new $\langle e', d', j_k \rangle$ if
     - it is not stored yet, and
     - $d' \leqslant B$
  5. Repeat from 2 until there are no more unmarked triples

Iterative procedure

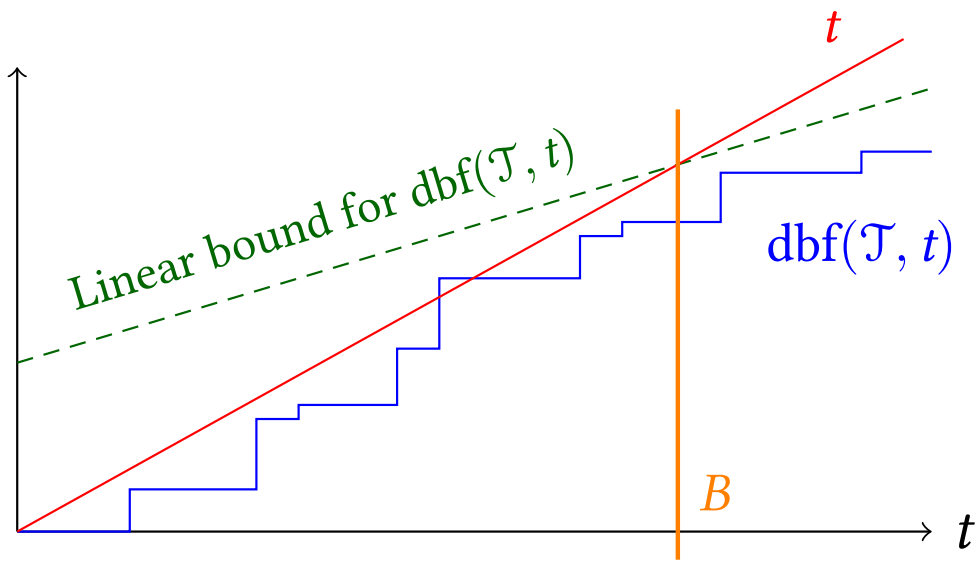
- Create all demand triples up to bound B:
  1. Store all *0-paths*, i.e., $\langle e(j), d(j), j \rangle$ for all vertices j
  2. Pick some stored unmarked demand triple $\langle e, d, j_i \rangle$, then *mark* it
  3. *Create new demand triples*:
     - For each successor vertex j_k of j_i
     - $e' = e + e(j_k)$
     - $d' = d - d(j_i) + p(j_i, j_k) + d(j_k)$
     - $\langle e', d', j_k \rangle$ is a new demand triple!
  4. Store each new $\langle e', d', j_k \rangle$ if
     - it is not stored yet, and
     - $d' \leqslant B$
  5. Repeat from 2 until there are no more unmarked triples
- More *efficient* procedure than enumerating all paths!
  - Note: Actual paths are never stored
  - Optimizations: Discard non-critical triples along the way

Iterative procedure

- Create all demand triples up to bound B:
  1. Store all *0-paths*, i.e., $\langle e(j), d(j), j \rangle$ for all vertices j
  2. Pick some stored unmarked demand triple $\langle e, d, j_i \rangle$, then *mark* it
  3. *Create new demand triples*:
     - For each successor vertex j_k of j_i
     - $e' = e + e(j_k)$
     - $d' = d - d(j_i) + p(j_i, j_k) + d(j_k)$
     - $\langle e', d', j_k \rangle$ is a new demand triple!
  4. Store each new $\langle e', d', j_k \rangle$ if
     - it is not stored yet, and
     - $d' \leqslant B$
  5. Repeat from 2 until there are no more unmarked triples
- More *efficient* procedure than enumerating all paths!
  - Note: Actual paths are never stored
  - Optimizations: Discard non-critical triples along the way
- Exercise: What's $\mathrm{dbf}(\tau_i, 26)$ if τ_i is the task on previous slide?

Which t to check?

- Recall: Want to check $\mathrm{dbf}(\mathcal{T}, t) \leqslant t$ for all t
- Find a bound for t just like for sporadic tasks!

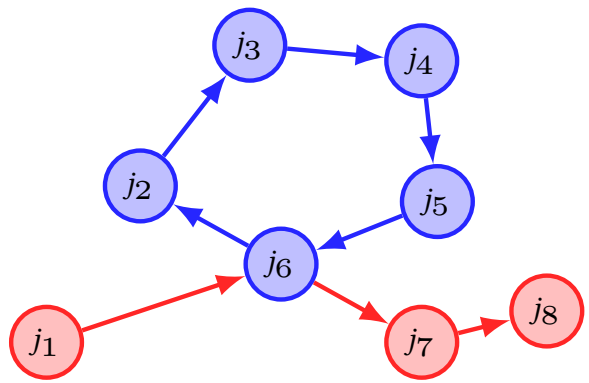


$$\forall t : \mathrm{dbf}(\mathcal{T}, t) \leqslant t$$

- Derive linear bound for $\mathrm{dbf}(\mathcal{T}, t)$
  - Intersection with t gives bound B
  - How to find the linear bound?

Linear bound for dbf

- Bound is based on *utilization*
  - Long-term demand "rate" (asymptotic)
  - For DRT: "most dense" cycle
  - Highest ratio execution demand (vertices) vs. duration (edges)
    - How to find this value? (Exercise!)



- Any path can be split into vertices in cycles and not in cycles
- Leads to

$$\mathrm{dbf}(\tau_i, t) \leqslant \mathrm{U}(\tau_i) \cdot t + \sum_k e(j_k)$$

- So, check $\mathrm{dbf}(\tau_i, t)$ for which t? (Exercise!)

DRT feasibility: Summary

- Feasibility test (or schedulability test for EDF), based on dbf
- First, compute the *utilization* for all tasks
  - Based on most dense cycles in graphs
- Derive *bound B*
- Compute $\mathrm{dbf}(\mathcal{T}, t)$ for all $t \leqslant B$
  - Uses iterative procedure with *demand triples*
  - Path abstraction to reduce complexity
- If $t \leqslant B$ with $\mathrm{dbf}(\mathcal{T}, t) > t$ is found, then $\mathcal{T}$ is infeasible
- Otherwise $\mathcal{T}$ is feasible

DRT feasibility: Summary

- Feasibility test (or schedulability test for EDF), based on dbf
- First, compute the *utilization* for all tasks
  - Based on most dense cycles in graphs
- Derive *bound B*
- Compute $\text{dbf}(\mathcal{T}, t)$ for all $t \leqslant B$
  - Uses iterative procedure with *demand triples*
  - Path abstraction to reduce complexity
- If $t \leqslant B$ with $\text{dbf}(\mathcal{T}, t) > t$ is found, then $\mathcal{T}$ is infeasible
- Otherwise $\mathcal{T}$ is feasible

- Read more in *The Digraph Real-Time Task Model* (Stigge et al., 2011)
- ...or in Martin Stigge's PhD thesis
- Play with a Python implementation: libdrt

Generalize further? Timed automata!

Idea

Annotate locations on timed automata with job parameters (WCET, relative deadline) let a job be created every time such a location is visited.

Analysis is generally costly, but sometimes fast enough.

Academic tool based on Uppaal: `http://www.timestool.com/`