A Grouping Approach to Task Scheduling with Functional and Non-Functional Requirements

Luca Santinelli, W. Puffitsch, Arnaud Dumerat, Frédéric Boniol, Claire Pagetti, Victor Jegu

A Grouping Approach to Task Scheduling with Functional and Non-Functional Requirements

Luca Santinelli[1], Wolfgang Puffitsch[2], Arnaud Dumerat[3], Frederic Boniol[1], Claire Pagetti[1], and Jegu Victor[3]

[1]ONERA Toulouse, name.surname@onera.fr

[2]DTU Compute Copenhagen, wopu@dtu.dk

[3]AIRBUS Toulouse, ename.surname@airbus.com

Abstract

The problem of finding a feasible task scheduling with both functional and non-functional requirements has risen to complexities never experienced before. In this paper we propose a functional task classification stage through which tasks are grouped together according to their functional properties. Then the task scheduling problem is reduced into a group scheduling problem where inter-group and intra-group ordering are experienced to cope with the timing requirements systems demand. In order to quantify the effects of the proposed grouping approach, we evaluate the approach by applying it to several realistic case studies.

1 Introduction

The well-known definition of real-time systems states that the correctness of such systems "depends not only on the logical result of the computation but also on the time at which the results are produced" [1]. Therefore, real-time systems combine functional and non-functional requirements. On the one hand, the result must be functionally correct. On the other hand, the result must arrive in time.

Reasoning about functional requirements can mostly abstract away implementation and run-time aspects. In contrast, non-functional requirements tend to depend heavily on such aspects. In this paper, we consider functional requirements on the task level, which translate to precedence constraints. A schedule that observes the functional dependencies between tasks will produce a functionally correct result, but can fail miserably when considering the time at which results arrive. Moreover, execution times of tasks are irrelevant with regard to functional requirements, but determine if a schedule that fulfills the non-functional requirements can exist at all.

Finding valid schedules can become problematic for embedded systems with large task sets that include both functional requirements in the form of dependencies between tasks and non-functional requirements in terms of timing constraints. On the one hand, the dependencies defeat traditional schedulability tests that assume independence between tasks. On the other hand, exact approaches face limitations when being applied to task sets with thousands of tasks [2, 3, 4].

In order to make such task sets tractable, it is necessary to reduce the complexity of the scheduling problem. We propose to group tasks according to their functional requirements, and to perform scheduling on these groups of tasks rather than individual tasks. This would reduces the number of items to be scheduled and consequently the complexity of the scheduling problem to be solved.

Contributions: This paper investigates the use of the functional requirements to group tasks and reduce the complexity of the scheduling problem. The goal is to find an abstraction level that combines both the functional and the non-functional view of a system, while being sufficiently abstract to enable the treatment of large task sets. In particular, we consider the effects of different abstractions (we call them grouping) policies on non-functional properties such as latencies.

Organization of the paper: In Section 2 the scheduling problem is presented together with the graph representation to the task functional requirements. Section 3 defines the grouping idea as well as two grouping policies proposed, while Section 4 considers the different task orderings with or without grouping. Section 5 investigates the effect of task grouping to timing constraints in the form of latency constraints. In Section 6 we apply our grouping framework to avionic examples with multiple tasks functionally connected. The grouping strategy is compared with the non-grouping. Finally, Section 7 concludes the paper and provides future perspectives for the presented work.

1.1 Related Work

There are the several works on combining both functional requirements and timing constraints to the real-time scheduling of tasks. We have been inspired by them to develop our grouping framework.

Chetto et al. [5] first considered the effect of precedence constraints between tasks on the dynamic priority scheduling problem. That paper proposes an algorithm to accept or reject aperiodic tasks with precedence constraints to guarantee the timing behavior of the rest of the system's tasks.

Spuri et al. [6] extended formal results on precedence constrained tasks to arbitrarily timed tasks with preemption; the precedence constraints are enforced through dynamic priority schedulers. Goddard et al. [7, 8] refer to data flows to exploit real-time properties such as latencies and buffers and perform schedulability analyses.

Cucu-Grosjean et al. have tackled with the scheduling problem and graph representations for the functional constraints [9, 10]. That paper models scheduling problems with precedence, periodicity and latency constraints. Yomsi and Sorel [11] have instead considered non-schedulability conditions to precedence constrained tasks and restrict the study to only potentially schedulable systems.

All those works are inspired, among the others, by Clark [12] which outlines the task dependency problem together with the complexity of the scheduling problem when the dependences are dynamic, and the work of Natale et al. [13] where end-to-end timing constraints are applied to guarantee timing constraints of distributed applications communicating via synchronous primitives. The joint and coordinated scheduling of tasks and messages is defined, while precedence constraints are converted into pseudo-deadline. In there, a combination of off-line and on-line scheduling is proposed. To the best of our knowledge, the approaches developed so far do not rely on task classification (task grouping) techniques to reduce the number of objects which have to be scheduled while guaranteeing both functional and non-functional constraints. With this work, we continue the exploration of the grouping approach started with [14].

There are several exact (i.e., non-heuristic) approaches to finding off-line schedules for task sets with precedence constraints. An early approach is the work by Xu and Parnas [2], which starts from a heuristic solution that is consistent with the functional requirements and then uses a branch-and-bound algorithm to improve the solution with regard to the non-functional requirements. The branch-and-bound algorithm continues until it finds a schedule that fulfills both the functional and non-functional requirements or until it has proven that no valid schedule exists.

Other approaches combine the functional and non-functional requirements and try to find a solution that satisfies both in a single step. Grolleau and Choquet-Geniet use Petri nets to model the scheduling of dependent tasks [3].

Ekelin [4] explores the use of constraint programming to solve scheduling problems, and presents several optimizations to speed up the search for a valid solution. Priced timed automata have also been proposed to schedule real-time tasks [15]. Their flexibility simplifies the adaption of different task and execution models.

While these approaches can be fairly efficient for some cases, they cannot escape the fact that the underlying problem is a hard problem. As the number of tasks to be scheduled increases, finding a valid schedule becomes unfeasible rather quickly. Reducing the number of items to be scheduled, as our grouping approach proposes, helps in making real-time systems with thousands of tasks tractable.

2 Problem Statement & Backgrounding

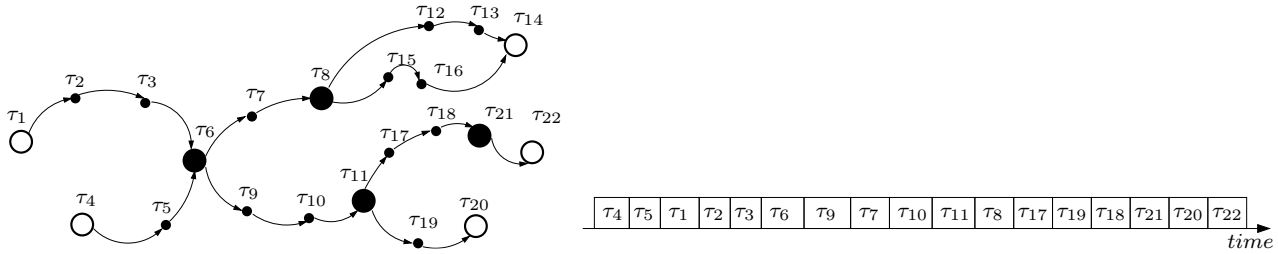
In real-time systems, it is necessary to schedule the execution of tasks such that the system meets its timing constraints. In addition to these non-functional requirements, the potential orderings of tasks can also be constrained by functional requirements, i.e., precedence constraints, equivalently named functional dependences. Today's real-time systems comprise a large number of tasks, timing constraints and precedence constraints and their size is expected to grow further. Consequently, the complexity to find a schedule that fulfills all constraints becomes increasingly harder, in particular because the underlying problem is NP-hard [13, 16].

In this work we propose a two-stage approach to tackle the real-time scheduling problem. The framework first classifies tasks into groups according to their functional requirements, and then schedule the groups of tasks to guarantee the timing requirements. Our grouping scheduling approach is developed assuming a) data dependencies as precedence constraints with a graph description, b) timing constraints described in terms of end-to-end latencies, although [10] extended the scheduling problem with precedence constraints to deadline timing constraints, c) off-line, non-preemptive and single-processor scheduling, d) mono-rate with synchronous tasks and the same period.

Example 2.1. *Figure 1 shows the reference example we apply to explain the approach under development. The* 22 *tasks have functional requirements described in Figure 1(a) with the precedence dependences as edges of the graph. Figure 1(b) shows an example of task scheduling compliant to the functional requirements.*

2.1 Functional Backgrounding

A real-time system can be seen as a task set $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ where each composing task τ_j is described by a tuple (a_j, C_j, T_j) with C_j being the task worst-case execution time, T_j the inter-arrival time of the task instances (jobs) and



(a) Functional dependencies through data flow and graph representation

(b) An example of a possible task execution ordering for the tasks within the graph

Figure 1: Task functional dependence description and scheduling.

a_j the task activation which repeats at any of its instance. All the tasks of Γ are assumed with the same period to approach safety-critical avionic platforms. Recent works have shown that in safety-critical avionic platforms the strict time-oriented approach is applied with mono and multiple-rate tasks, [17, 18]. In our first stage we start with the mono-rate case to converge to the multiple-rate case in the future. Thus all the tasks share the same period T_j.

In our model, the precedence constraints (i.e. data dependences) between tasks are described as a directed acyclic graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V}$ is the set of tasks Γ, $\Gamma \equiv \mathbb{V}$, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the set of edges which represents the precedence constraints between tasks.

Although classical real-time system models assume task execution as an infinite sequence of tasks instances (jobs), we keep our model within the finite directed graph scenario considering the tasks as all tasks have the same period and there are no offsets, jobs and tasks are equivalent.

We model the task set as a graph, where tasks are represented by vertices and precedence constraints by edges.

Definition 2.2 (Precedence Constraint)**.** *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, an edge between two tasks τ_j and τ_k, $\tau_j \rightarrow \tau_k$, represents a precedence constraint between τ_j and τ_k.*

A path $p(\tau_i, \tau_o)$ from task τ_i to task τ_o within a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ is an alternating sequence $\langle \tau_i, \tau_i \rightarrow \tau_1, \tau_1, \ldots, \tau_n \rightarrow \tau_o, \tau_o \rangle$ of vertices and distinct edges where $\tau_i, \tau_1, \ldots, \tau_n, \tau_o \in \mathbb{V}$ and $\tau_i \rightarrow \tau_1, \ldots, \tau_n \rightarrow \tau_o \in \mathbb{E}$. We denote by $\mathbb{P}$ the set of all the paths in $\mathbb{G}$ and we say that two tasks τ_k and τ_j are *connected* if there is at least one path $p(\tau_k, \tau_j) \in P$. As the number of vertices and edges is finite and the graph is acyclic, P is a finite set. By $P(\tau_k, \tau_j)$ we denote the set of all the paths from τ_k to τ_j; $M(\tau_k, \tau_j)$ instead is the set of tasks belonging to all the possible paths from τ_k to τ_j.

Definition 2.3 (Predecessors and Successors)**.** *The set of predecessors (successors) of a task τ_i is denoted by $preds(\tau_i)$ ($succs(\tau_i)$) and can be defined as $preds(\tau_j) \stackrel{def}{=} \{\tau_k \mid \tau_k \rightarrow \tau_j\}$ and $succs(\tau_j) \stackrel{def}{=} \{\tau_k \mid \tau_j \rightarrow \tau_k\}$.*

The cardinality of a task is the sum of the number of predecessors and successors, $\|\tau_k\| = \|preds(\tau_k)\| + \|succs(\tau_k)\|$.

Within a functional graph, we can classify the tasks according to the structure of the graph. We have "begin tasks" (deb) as those with no predecessors, "intermediate tasks" (int) with both predecessors and successors, and "end tasks" or leaves ($leaf$) with just predecessor tasks.

The functional requirements results into functional dependences between tasks.

Definition 2.4 (Direct Dependence $\succ$)**.** *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, a task τ_k directly depends on a task τ_j, $\tau_k \succ \tau_j$ if $\tau_j \rightarrow \tau_k$.*

The direct dependence relation can be extended transitively to a notion of *functional dependence*.

Definition 2.5 (Functional Dependence $\rhd$)**.** *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, a task τ_k* functionally depends *on a task τ_j, $\tau_k \rhd \tau_j$ if there exists at least one path $p(\tau_k, \tau_j) \in \mathbb{P}$ connecting τ_k and τ_j within $\mathbb{G}$.*

The functional dependence is a transitive relation between tasks; indeed, if $\tau_k \rightarrow \tau_j$ and $\tau_j \rightarrow \tau_r$, then $\tau_k \rhd \tau_r$. Thus the direct dependence is a stricter definition of dependence $\succ \subseteq \rhd$, which does not include the transitivity property.

Definition 2.6 (Functional Independence $\overline{\rhd}$)**.** *The notion of independence is the opposite (the negation) of the dependence, $\overline{\rhd}$. Two tasks τ_j and τ_k are called independent, $\tau_j \overline{\rhd} \tau_k$ if $\nexists\, p(\tau_j, \tau_k) \in \mathbb{P}$.*

The functional description implies a partial ordering between tasks, but (usually) allows more than one valid schedule. We can define equivalence among scheduling as follows: *two scheduling resulting from the same partial ordering are functionally equivalent.* The objective of the scheduling problem is to find in a total ordering of task such that complies to both functional and non-functional requirements.

3 Functional Grouping

To simplify the scheduling problem with functional dependencies we propose to classify tasks according to their dependencies and create groups of tasks.

Definition 3.1 (Grouping)**.** *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, a grouping $\mathcal{G} = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ divides the task set into disjoint subsets such that $\forall \mathcal{G}_i, \mathcal{G}_j \in \mathcal{G},\ \mathcal{G}_i \cap \mathcal{G}_j = \emptyset$ and $\bigcup_{i=1}^{n} \mathcal{G}_i = \mathbb{V}$.*

The notion of direct and functional dependence can be extended from tasks to groups of tasks by considering groups dependent if there is a dependence between any of their tasks:

$$\mathcal{G}_i \succ \mathcal{G}_k \quad \Leftrightarrow \quad \exists \tau_j \in \mathcal{G}_i, \tau_l \in \mathcal{G}_k, \tau_j \succ \tau_l \tag{1}$$

$$\mathcal{G}_i \rhd \mathcal{G}_k \quad \Leftrightarrow \quad \exists \tau_j \in \mathcal{G}_i, \tau_l \in \mathcal{G}_k, \tau_j \rhd \tau_l \tag{2}$$

The criteria for grouping may be chosen arbitrarily, and some grouping are more helpful with regard to scheduling than others. In the following, we focus on two classes of grouping:

- *Independence grouping*, which exploits the notion of independence to partition the task set.
- *Dependence grouping*, which creates groups of dependence tasks.

3.1 Independence Grouping

Definition 3.2 (Independence Grouping)**.** *We call a grouping $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots\}$ an independence grouping if only independent tasks belong to the same partition $\mathcal{I}_i$,*

$$\forall\ \mathcal{I}_i \in \mathcal{I}, \quad \forall\ \tau_j, \tau_k \in \mathcal{I}_i, \quad \tau_j \overline{\rhd} \tau_k. \tag{3}$$

An independence grouping $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots\}$ partitions the task set into groups $\mathcal{I}_i$, where all the tasks within are independent among each others.

This independence definition does not infer a unique grouping. A trivial independence grouping would be a grouping where each partition contains exactly one task. Obviously, such a grouping would not be particularly helpful. However, even when considering non-trivial cases, grouping according to independence allows for some ambiguity. Consider tasks τ_2, τ_3 and τ_5 in the example given in Figure 1(a). Tasks τ_2 and τ_5 are independent, as well as tasks τ_3 and τ_5. In contrast, tasks τ_2 and τ_3 are dependent and cannot belong to the same independence group. Therefore, we can group such that either $\{\tau_2, \tau_5\} \in \mathcal{I}$ or $\{\tau_3, \tau_5\} \in \mathcal{I}$. None of these alternatives is inherently better than the other.

In the following, we present two algorithms to create independence groupings. The first algorithm moves "forward", from begin tasks towards leaf tasks, while the second one moves "backward" in the task graph.

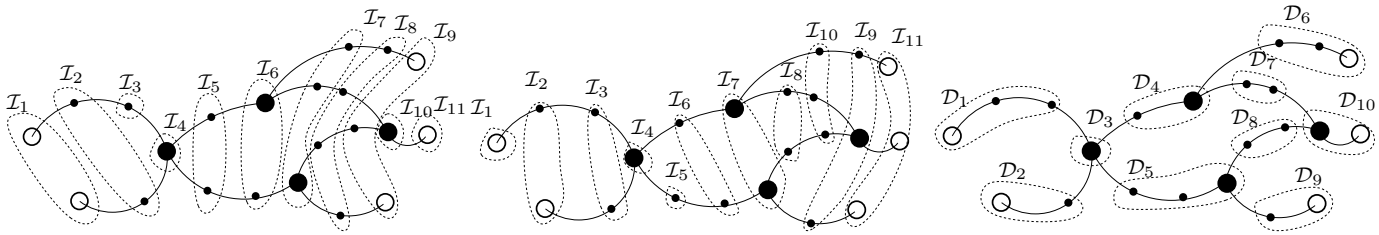
3.1.1 Forward independence grouping

Algorithm 1 shows an algorithm to create an independence grouping by moving forward through the task graph. In the first step, it puts all tasks without predecessor tasks (begin tasks) into the same group. As none of these tasks have predecessors, it cannot be the case that $\tau_i \rhd \tau_k$ and the group consists only of independent tasks. The algorithm then removes these nodes and the related edges from the graph and creates a group which contains tasks without predecessors in the new task graph. The same reasoning as before applies, and all tasks in the group are independent. While removing tasks from the graph no relevant edges are removed; only the precedent tasks are evicted and the dependences among remaining tasks remain unaffected. Therefore, Algorithm 1 creates a valid independence grouping.

3.1.2 Backward independence grouping

The inverse to Algorithm 1 is backward independence grouping, which starts from tasks without successors and moves towards the input nodes. Its implementation is shown in Algorithm 2. The reasoning to show that the algorithm creates and independence grouping is analogous to the reasoning for the forward algorithm.

Example 3.3. *Figure 2(a) outlines the result of applying the forward independence algorithm to Example 1; the algorithm creates* 11 *groups where, for example, τ_2 and τ_5 while leaving task τ_3 in a group of its own. Applying the backward algorithm to the example leads to a different grouping, which is shown in Figure 2(a). In contrast to the grouping created by forward algorithm, τ_3 and τ_5 are now in the same group. Although resulting group partitions are different, we expect both algorithms to result in similar complexity reductions for the scheduling problem.*



(a) Independence grouping with the forward algorithm, Algorithm 1

(b) Independence grouping with the backward algorithm, Algorithm 2

(c) Dependence grouping, Algorithm 3

Figure 2: Functional grouping.

3.2 Dependence Grouping

Instead of grouping independent tasks, we can group tasks that form chains of dependent tasks.

Definition 3.4 (Chain)**.** *Two tasks τ_i and τ_k form a chain if τ_k is the only successor of τ_i and τ_i the only predecessor of τ_k, or if there exists a sequence of chains between τ_i and τ_k through intermediate tasks.*

$$\tau_i \rightsquigarrow \tau_k \stackrel{def}{=} (succs(\tau_i) = \{\tau_k\} \wedge preds(\tau_k) = \{\tau_i\}) \vee (\exists \tau_l, \tau_i \rightsquigarrow \tau_l \wedge \tau_l \rightsquigarrow \tau_k) \tag{4}$$

Definition 3.5 (Dependence Grouping)**.** *A dependence grouping $\mathcal{D}$ is a partitioning of the task set such that chains of dependent tasks belong to the same group $\mathcal{D}_i$,*

$$\forall \mathcal{D}_i \in \mathcal{D}, \quad \forall \tau_j, \tau_k \in \mathcal{D}_i, \ \textit{with } j \neq k \quad \Leftrightarrow \quad \tau_j \rightsquigarrow \tau_k \vee \tau_k \rightsquigarrow \tau_j. \tag{5}$$

A dependence grouping $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots\}$ partitions the task set into groups $\mathcal{D}_i$, where all the tasks within are dependent among each others.

Algorithm 3 computes the dependence grouping. It starts from the tasks without any predecessors and iterates over them creating a group for each of these, which includes the task and all those that form a chain with it. Afterwards, it removes the tasks that already are part of a group and continues until the task graph is empty.

Algorithm 1 Forward independence grouping algorithm

Input: Γ
Output: $\mathcal{I}_i$
1: $i \leftarrow 1$, $\mathcal{R} \leftarrow \Gamma$
2: **while** $\mathcal{R} \neq \emptyset$ **do**
3: $\quad \mathcal{I}_i \leftarrow \{\tau_j \mid preds(\mathcal{R}, \tau_j) = \emptyset\}$
4: $\quad \mathcal{R} \leftarrow \mathcal{R} - \mathcal{I}_i$
5: $\quad i \leftarrow i + 1$
6: **end while**

Algorithm 2 Backward independence grouping algorithm

Input: Γ
Output: $\mathcal{I}_i$
1: $i \leftarrow 1$, $\mathcal{R} \leftarrow \Gamma$
2: **while** $\mathcal{R} \neq \emptyset$ **do**
3: $\quad \mathcal{I}_i \leftarrow \{\tau_j \mid succs(\mathcal{R}, \tau_j) = \emptyset\}$
4: $\quad \mathcal{R} \leftarrow \mathcal{R} - \mathcal{I}_i$
5: $\quad i \leftarrow i + 1$
6: **end while**

Algorithm 3 Dependence grouping algorithm

Input: Γ
Output: $\mathcal{D}_i$
1: $i \leftarrow 1$, $\mathcal{R} \leftarrow \Gamma$
2: **while** $\mathcal{R} \neq \emptyset$ **do**
3: $\quad T \leftarrow \{\tau_i \mid preds(\mathcal{R}, \tau_i) = \emptyset\}$
4: $\quad$ **for** $\tau_j \in T$ **do**
5: $\quad\quad \mathcal{D}_i \leftarrow \{\tau_j\} \cup \{\tau_k \mid \tau_j \rightsquigarrow \tau_k\}$
6: $\quad\quad i \leftarrow i + 1$
7: $\quad$ **end for**
8: $\quad \mathcal{R} \leftarrow \mathcal{R} - \bigcup_{j=1}^{i} \mathcal{D}_j$
9: **end while**

Algorithm 3 implements a *correct* dependence grouping since it explores the whole graph for the largest dependence groups, Equation (5). Intuitively, we can say that Algorithm 3 guarantees the larges dependence groups. In that sense it dominates all the possible dependence grouping which start from intermediate or leaf tasks. In the future we will formally prove the dominance.

Example 3.6. *Figure 2(c) shows the result of applying dependence grouping to Example 1 with* 10 *resulting groups, $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, $\mathcal{D}_4$, $\mathcal{D}_5$, $\mathcal{D}_6$, $\mathcal{D}_7$, $\mathcal{D}_8$, $\mathcal{D}_9$ and $\mathcal{D}_{10}$.*

4 Task Scheduling

The precedence constraints define a partial order for the activation times of tasks. For tasks τ_i and τ_k with $\tau_i \rhd \tau_k$, it must be the case that τ_i activates earlier than τ_k. For independent tasks, with $\tau_i \overline{\rhd} \tau_k$, the precedence constraints do not imply such an ordering among task activations. This means that the two scheduling where either τ_i executes before τ_j or τ_j executes before τ_i are equivalent from the functional point of view.

The partial order established by the functional dependences is transformed into a total order by the scheduling.

For a system of tasks Γ, a schedule S is a totally ordered set of activation times of all the tasks $S = \{a_j \in \mathbb{N}\}, \tau_j \in \mathbb{V}$ such that all the precedence constraints are satisfied [9]. $\mathbb{S}$ is the set of all the possible schedules.

With the group decomposition we are proposing, for a scheduling problem we can differentiate two "levels" of scheduling:

- the *inter-group* scheduling where it is tackled with group ordering. The groups are ordered according to functional and non-functional requirements.
- The *intra-group* scheduling, as the ordering of the tasks composing each group. The tasks within each group needs to be ordered facing both functional and non-functional requirements.

With an independence or a dependence grouping just a scheduling level (either intra-grouping or inter-grouping) needs to be approached to define the task total ordering. This separation reduces the complexity of the scheduling problem.

Example 4.1. *Taking the reference example from Figure 1, we have that the first block of the graph composed by the tasks* $\tau_1, \tau_2, \tau_3, \tau_4$ *and* τ_5 *can results into* 10 *possible schedules:*

$$\tau_1-\tau_2-\tau_3-\tau_4-\tau_5, \quad \tau_1-\tau_2-\tau_4-\tau_5-\tau_3, \quad \tau_1-\tau_4-\tau_5-\tau_2-\tau_3, \quad \tau_1-\tau_4-\tau_2-\tau_5-\tau_3,$$
$$\tau_1-\tau_4-\tau_2-\tau_3-\tau_5, \quad \tau_1-\tau_2-\tau_4-\tau_3-\tau_5, \quad \tau_4-\tau_5-\tau_1-\tau_2-\tau_3, \quad \tau_4-\tau_1-\tau_5-\tau_2-\tau_3,$$
$$\tau_4-\tau_1-\tau_2-\tau_5-\tau_3, \quad \tau_4-\tau_1-\tau_2-\tau_3-\tau_5.$$

All of them are compliant to the functional requirements, although they differ in terms timing.

Independence Scheduling With an independence grouping as computed by Algorithm 1 or Algorithm 2, the scheduling of groups is already fixed. The algorithms create groups greedily, such that groups must be executed in the same order as they were created for the forward algorithm, and in reverse order for the backward algorithm. However, as all tasks are independent, the execution order of tasks within a group can be chosen freely. The independence grouping reduces the possible task schedules, which in the worst case could eliminate all schedules that would satisfy the timing requirements. However, the grouping does not add any additional schedules, and is therefore safe with regard to both functional and non-functional requirements.

Example 4.2. *From Example 2.1 with a forward independence grouping it is* $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_5, \mathcal{I}_6, \mathcal{I}_7, \mathcal{I}_8, \mathcal{I}_9, \mathcal{I}_{10}, \mathcal{I}_{11}\}$ *where the group ordering is already decided as* $\mathcal{I}_1-\mathcal{I}_2-\mathcal{I}_3-\mathcal{I}_4-\mathcal{I}_5-\mathcal{I}_6-\mathcal{I}_7-\mathcal{I}_8-\mathcal{I}_9-\mathcal{I}_{10}-\mathcal{I}_{11}$*, Figure 2(a). The total ordering is obtained selecting for each set* $\mathcal{I}_i$ *an order for the composing tasks. This results into* $2!2!1!1!2!2!3!4!3!1!1! = 13824$ *possible task combinations tough, still a complex problem.*

Considering the subgraph $\mathcal{I}_1-\mathcal{I}_2-\mathcal{I}_3$ *from a forward independence grouping, there are* 4 *possible orderings with independence grouping* $\tau_1-\tau_4-\tau_2-\tau_5-\tau_3, \tau_1-\tau_4-\tau_5-\tau_2-\tau_3, \tau_4-\tau_1-\tau_2-\tau_5-\tau_3, \tau_4-\tau_1-\tau_5-\tau_2-\tau_3$*, out of the* 10 *possible without any grouping applied, Example 4.1. This is the degree of flexibility we lose by applying grouping while consistently reducing the complexity of the problem.*

With a backward independence grouping, Algorithm 2 and Figure 2(b) it would be $1!2!2!1!2!2!2!3!3!3! = 6912$*, which are slightly fewer combinations for possible schedules than for forward grouping. Considering the subgraph* $\mathcal{I}_1-\mathcal{I}_2-\mathcal{I}_3$ *from a backward independence grouping, there are* 4 *possible orderings with independence grouping* $\tau_1-\tau_4-\tau_2-\tau_5-\tau_3, \tau_1-\tau_4-\tau_2-\tau_3-\tau_5, \tau_1-\tau_2-\tau_4-\tau_5-\tau_3, \tau_1-\tau_2-\tau_4-\tau_3-\tau_5$*, out of the* 10 *possible without any grouping applied, Example 4.1. This is the degree of flexibility we lose and the difference with respect to a forward independence grouping.*

Dependence Scheduling With the dependence grouping paradigm, the ordering of tasks within a group is fixed by the functional constraints which are accounted by the grouping policy itself. In contrast, the scheduler has to find a suitable ordering of the groups of dependent tasks. Within this paradigm, the scheduling remains a mixture of functional and execution ordering with less flexibilities compared to the case without grouping. Like independence grouping, dependence grouping is safe with regard to the functional requirements, but may eliminate all schedules that would satisfy the timing constraints.

Example 4.3. *The resulting dependence scheduling for Example 2.1 derives from the following set of combinations* $\{\mathcal{D}_1, \mathcal{D}_2\}-\mathcal{D}_3-\{\mathcal{D}_4, \mathcal{D}_5\}-\{\mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_8, \mathcal{D}_9\}-\mathcal{D}_{10}$*, where the relative order between* $\mathcal{D}_1$ *and* $\mathcal{D}_2$ *does not affect the functional requirements as for the tuples* $\mathcal{D}_4$*,* $\mathcal{D}_5$*, and* $\mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_8$*,* $\mathcal{D}_9$*. The dependent scheduling results into* $2!1!2!4!1! = 96$ *possible ordering. Considering the partition* $\{\mathcal{D}_1, \mathcal{D}_2\}-\mathcal{D}_3$*, with a dependence grouping, we have* 2 *possible ordering depending on which between* $\mathcal{D}_1$ *or* $\mathcal{D}_2$ *is scheduled first, thus* $\tau_1-\tau_2-\tau_3-\tau_4-\tau_5, \tau_4-\tau_5-\tau_1-\tau_2-\tau_3$ *as a subset of the* 10 *possible without any grouping. We notice that with a dependence grouping we further lose scheduling possibilities in that particular configuration.*

5 Analysis with Timing Constraints

The next step of this work is the analysis of the grouping effect to the timing constraints. In particular, we consider latency constraints as end-to-end timing constraints to the task executions.

For a pair of tasks (τ_i, τ_o) belonging to a system and a schedule S of this system, we call latency the time between the start of the task τ_i and the end of a task τ_o. We denote this time by $L(\tau_i, \tau_o)$, as the input/output latency (I/O latency). With no idle time in S (the work-conserving case), it is $L(\tau_i, \tau_o) = \sum_{\forall c \mid a_i \leq a_c \leq a_o} C_c$, where the activations a_i and a_o belong to the scheduling S. More generally, the latency is given by

$$L(\tau_i, \tau_o) = a_o + C_o - a_i, \tag{6}$$

when including possible idle times in the scheduling. The activation instants a encode the interference from other task executions. The latency constraint is when the latency is less than or equal to an imposed bound l.

$$L(\tau_i, \tau_o) \leq l_{L(\tau_i,\tau_o)} \quad \Leftrightarrow \quad a_o + C_o - a_i \leq l_{L(\tau_i,\tau_o)} \tag{7}$$

The I/O latencies characterizing a task set Γ, described as Equation (7), are grouped into the set $\mathbb{L}$.

The functional and non-functional scheduling problem becomes exploring the possible task ordering S looking for those that satisfy both the functional requirements $\mathbb{G}$ and the latency constraints $\mathbb{L}$.

Part of the complexity of the scheduling problem comes from the non-trivial relation between the task ordering and the task activations. A latency $L(\tau_i, \tau_o)$ is not only given by the tasks composing $P(\tau_i, \tau_o)$. There exist tasks τ_j such that although not in the input-output set of paths, $\tau_j \notin M(\tau_i, \tau_o)$, they have to be scheduled after τ_i and before τ_o, due to functional requirements. Those τ_j contributes to $L(\tau_i, \tau_o)$. As we will see, the grouping analysis we propose allows to extract simple relations between arrivals and latencies, thanks to the task classification and the group partitions.

With groups partitioning a functional graph, a latency can be decomposed into the groups in terms of their contributions to the latency itself. A group of tasks $\Omega_{L(\tau_i,\tau_o)}$ *affects* (equivalently *contributes* to), an I/O latency $L(\tau_i, \tau_o)$ if

$$\forall\ \tau_j \in \Omega_{L(\tau_i,\tau_o)} \quad \tau_j \in M(\tau_i, \tau_o) \quad \vee \quad \exists \tau_k \in M(\tau_i, \tau_o) \mid \tau_j \in preds(\tau_k). \tag{8}$$

$\Omega_{L(\tau_i,\tau_o)}$ is the set of all tasks which contributes to $L(\tau_i, \tau_o)$.

By extension, a partition $\mathcal{G}_i$ affects the latency $L(\tau_i, \tau_o)$ if $\exists \tau_j \in \mathcal{G}_i$ such that τ_j affects $L(\tau_i, \tau_o)$. Then, given $\mathcal{G}_{L(\tau_i,\tau_o)} = \{\mathcal{G}_{1,L(\tau_i,\tau_o)}, \ldots, \mathcal{G}_{n,L(\tau_i,\tau_o)}\} \subseteq \mathcal{G}$ the subset of partitions affecting the latency $L(\tau_i, \tau_o)$, it is possible to write $L(\tau_i, \tau_o)$ as the combination of partition contributions, $L_{\mathcal{G}_{1,L(\tau_i,\tau_o)}} + L_{\mathcal{G}_{2,L(\tau_i,\tau_o)}} + \ldots + L_{\mathcal{G}_{n,L(\tau_i,\tau_o)}}$. The latency constraint becomes

$$L_{\mathcal{G}_{1,L(\tau_i,\tau_o)}} + L_{\mathcal{G}_{2,L(\tau_i,\tau_o)}} + \ldots + L_{\mathcal{G}_{n,L(\tau_i,\tau_o)}} \leq l_{L(\tau_i,\tau_o)}. \tag{9}$$

5.1 Latencies with Independences

The independence grouping policy modifies I/O latency. With an independence task set partitioning $\mathcal{I}$, it is possible to identify the subset of independence groups affecting $L(\tau_i, \tau_o)$, $\mathcal{I}_{L(\tau_i,\tau_o)} = \{\mathcal{I}_{1,L(\tau_i,\tau_o)}, \mathcal{I}_{2,L(\tau_i,\tau_o)}, \ldots, \mathcal{I}_{n,L(\tau_i,\tau_o)}\}$, $\mathcal{I}_{L(\tau_i,\tau_o)} \subseteq \mathcal{I}$. With Equation (9) applied to independence groups, $L(\tau_i, \tau_o) \leq l_{L(\tau_i,\tau_o)}$ can decomposed into its contribution from $\mathcal{I}_{L(\tau_i,\tau_o)}$,

$$L_{\mathcal{I}_{1,L(\tau_i,\tau_o)}} + L_{\mathcal{I}_{2,L(\tau_i,\tau_o)}} + \ldots + L_{\mathcal{I}_{n,L(\tau_i,\tau_o)}} \leq l_{L(\tau_i,\tau_o)}. \tag{10}$$

The independence introduces pessimism in the end-to-end timing requirement which results into larger then or equal to latencies, $L_{\mathcal{I}_{1,L(\tau_i,\tau_o)}} + L_{\mathcal{I}_{2,L(\tau_i,\tau_o)}} + \ldots + L_{\mathcal{I}_{n,L(\tau_i,\tau_o)}} \leq l_{L(\tau_i,\tau_o)} \leq L(\tau_i, \tau_o)$.

Theorem 5.1 (Independence Scheduling)**.** *Given a task set $\Gamma = \{\ldots, \tau_i, \ldots, \tau_o, \ldots\}$ with precedences described by $\mathbb{G}$, partitioned into independence groups and constrained by I/O latencies $\mathbb{L}$, each of them ($L(\tau_i, \tau_o) \leq l_{L(\tau_i,\tau_o)}$) involving the independence groups $\{\mathcal{I}_{1,L(\tau_i,\tau_o)}, \ldots, \mathcal{I}_{n,L(\tau_i,\tau_o)}\}$. Γ is schedulable if $\forall\ L(\tau_i, \tau_o) \in \mathbb{L}$*

$$L_{\mathcal{I}_{1,L(\tau_i,\tau_o)}} + L_{\mathcal{I}_{2,L(\tau_i,\tau_o)}} + \ldots + L_{\mathcal{I}_{n,L(\tau_i,\tau_o)}} \leq l_{L(\tau_i,\tau_o)}. \tag{11}$$

Proof. Equation (10) applied to $\{\mathcal{I}_{1,L(\tau_i,\tau_o)}, \mathcal{I}_{2,L(\tau_i,\tau_o)}, \ldots \mathcal{I}_{n,L(\tau_i,\tau_o)}\}$ defines a compositional relationship to the I/O latency $L(\tau_i, \tau_o)$. A sufficient condition to the schedulability is that the latency bound is verified. Thus the theorem follow for every $L(\tau_i, \tau_o) \in \mathbb{L}$. □

Lemma 5.2 (Independence Latency)**.** *With and independence grouping $\mathcal{I}$ and for a latency $L(\tau_i, \tau_o)$, $\mathcal{I}_{L(\tau_i,\tau_o)} = \{\mathcal{I}_{1,L(\tau_i,\tau_o)}, \mathcal{I}_{2,L(\tau_i,\tau_o)}, \ldots, \mathcal{I}_{n,L(\tau_i,\tau_o)}\} \subseteq \mathcal{I}$ is the set of groups affecting $L(\tau_i, \tau_o)$. A sufficient condition to the latency constraints $L(\tau_i, \tau_o) \leq l_{L(\tau_i,\tau_o)}$ is that $l_{1,L(\tau_i,\tau_o)} + l_{2,L(\tau_i,\tau_o)} + \ldots + l_{n,L(\tau_i,\tau_o)} \leq l_{L(\tau_i,\tau_o)}$, with $l_{h,L(\tau_i,\tau_o)} = max_{(\tau_s,\tau_r) \in \mathcal{I}_{h,L(\tau_i,\tau_o)}} \{a_r + C_r - a_s\}$.*

Proof. It is possible to decompose $L(\tau_i, \tau_o)$ into the effects of the composing independence groups $\mathcal{I}_{h,L(\tau_i,\tau_o)}$, Equation (9). The worst-case latency contribution from each $\mathcal{I}_{h,L(\tau_i,\tau_o)} \in \mathcal{I}_{L(\tau_i,\tau_o)}$ is $l_{\mathcal{I}_{h,L(\tau_i,\tau_o)}} = max_{(\tau_s,\tau_r)\in\mathcal{I}_{h,L(\tau_i,\tau_o)}}\{a_r + C_r - a_s\}$. It is then $a_o + C_o - a_i \leq l_{1,L(\tau_i,\tau_o)} + l_{2,L(\tau_i,\tau_o)} + \ldots + l_{n,L(\tau_i,\tau_o)}$, and the latency condition can be verified if $a_o + C_o - a_i \leq l_{1,L(\tau_i,\tau_o)} + l_{2,L(\tau_i,\tau_o)} + \ldots + l_{n,L(\tau_i,\tau_o)} \leq l_{L(\tau_i,\tau_o)}$. The lemma comes as an upper bound to the group latency constraint. □

In terms of contributions to the latency we can differentiate independence groups according to their role with the I/O latency itself. For each $L(\tau_i, \tau_o)$ there is a) an *input group* $\mathcal{I}_{I,L(\tau_i,\tau_o)}$, containing the latency input task τ_i, b) an *output group* $\mathcal{I}_{O,L(\tau_i,\tau_o)}$, as the group which includes the output task τ_o of the latency, c) *intermediate groups* $\mathcal{I}_{int,L(\tau_i,\tau_o)}$, as those groups which have tasks belonging to the I/O latency but neither the input task nor the output task. Each group contributes differently to the latency.

Inputs Given the input task τ_i for the latency constraint $L(\tau_i, \tau_o) \leq l_{L(\tau_i,\tau_o)}$, the latency for the input group $\mathcal{I}_{I,L(\tau_i,\tau_o)}$ depends on the task activation a_i which is driven by the tasks belonging to $\mathcal{I}_{I,L(\tau_i,\tau_o)}$ and executed before τ_i. With $C_{I,L(\tau_i,\tau_o)}$ the total computation time of $\mathcal{I}_{I,L(\tau_i,\tau_o)}$, $C_{I,L(\tau_i,\tau_o)} = \sum_{\tau_h\in\mathcal{I}_{I,L(\tau_i,\tau_o)}} C_h$ (considering tasks inside a group executed consecutively), the latency of that group is $L_{I,L(\tau_i,\tau_o)} = a_{I,L(\tau_i,\tau_o)} + C_{I,L(\tau_i,\tau_o)} - a_i$ with $a_{I,L(\tau_i,\tau_o)}$ the activation of the first task of $\mathcal{I}_{I,L(\tau_i,\tau_o)}$ scheduled. Thus the latency constraint

$$a_{I,L(\tau_i,\tau_o)} + C_{I,L(\tau_i,\tau_o)} - a_i \leq l_{I,L(\tau_i,\tau_o)}, \tag{12}$$

where the bound $l_{I,L(\tau_i,\tau_o)}$ could be computed as in Lemma 5.2 or imposed. In its generic form, $C_{I,L(\tau_i,\tau_o)}$ can include the activation of the tasks, but both $C_{I,L(\tau_i,\tau_o)}$ and $a_{I,L(\tau_i,\tau_o)}$ are fixed and representable as $ET_{I,L(\tau_i,\tau_o)} = C_{I,L(\tau_i,\tau_o)} + a_{I,L(\tau_i,\tau_o)}$. The only parameter that can change is the task τ_i activation.

$$a_i \stackrel{def}{=} \begin{cases} 0 & \text{if } \tau_i \text{ scheduled first} \\ \sum_{\tau_h\in\mathcal{I}_{I,L(\tau_i,\tau_o)}\setminus\{\tau_i\} \mid (\tau_h \text{ before } \tau_i)} C_h + \delta_{I,L(\tau_i,\tau_o)} & \text{otherwise,} \end{cases} \tag{13}$$

with $\delta_{I,L(\tau_i,\tau_o)}$ the time due to possible intermediate task activations. It includes in the latency definition the non-consecutive task execution: the idle time which could exist. $a_i \geq \sum_{\tau_h\in\mathcal{I}_{I,L(\tau_i,\tau_o)}\setminus\{\tau_i\} \mid \tau_h \text{ executed before } \tau_i} C_h$. Equation (12) gives a bound to τ_i activations, $a_i \geq ET_{I,L(\tau_i,\tau_o)} - l_{I,L(\tau_i,\tau_o)}$ which limits the activation time of τ_i for a feasible intra-group ordering: a) if $ET_{I,L(\tau_i,\tau_o)} \leq l_{I,L(\tau_i,\tau_o)}$ then a_i does not have constraints, b) else, $ET_{I,L(\tau_i,\tau_o)} > l_{I,L(\tau_i,\tau_o)}$, a_i have to be larger or equal to $ET_{I,L(\tau_i,\tau_o)} - l_{I,L(\tau_i,\tau_o)}$. Playing with the intra-group task ordering it is possible to find the scheduling which optimize both the single latency constraint $L(\tau_i, \tau_o)$ (trivial problem) and the whole latency constraint set $\mathbb{L}$.

Outputs Given an output task τ_o for the latency $L(\tau_i, \tau_o)$, within the output group $\mathcal{I}_{O,L(\tau_i,\tau_o)}$ the latency is $L_{\mathcal{I}_{O,L(\tau_i,\tau_o)}} = a_o + C_o$. The contribution to the activation of τ_o is limited to the tasks belonging to $\mathcal{I}_{O,L(\tau_i,\tau_o)}$ executed before τ_o,

$$a_o \stackrel{def}{=} \begin{cases} 0 & \text{if } \tau_o \text{ scheduled first} \\ \sum_{\tau_h\in\mathcal{I}_{k,L(\tau_i,\tau_o)}\setminus\{\tau_o\} \mid (\tau_h \text{ before } \tau_o)} C_h + \delta_{O,L(\tau_i,\tau_o)} & \text{otherwise.} \end{cases} \tag{14}$$

$\delta_{I,L(\tau_i,\tau_o)}$ is to take into account idle time on the scheduling of $\mathcal{I}_{O,L(\tau_i,\tau_o)}$ tasks. It is $a_o \geq \sum_{\tau_h\in\mathcal{I}_{k,L(\tau_i,\tau_o)}\setminus\{\tau_o\} \mid \tau_h \text{ before } \tau_o} C_h$.

Input-Outputs A particolar condition is when a group contains both the input and the output tasks of a latency $L(\tau_i, \tau_o)$; we call it input-output group $\mathcal{I}_{IO,L(\tau_i,\tau_o)}$. In this case, $\mathcal{I}_{IO,L(\tau_i,\tau_o)}$ is the only group affecting $L(\tau_i, \tau_o)$. The latency contribution of $\mathcal{I}_{IO,L(\tau_i,\tau_o)}$ (which is the only contribution) depends on the relative task execution order within it, in particular from those tasks executed after τ_i and before τ_o.

$$a_oC_o - a_i \stackrel{def}{=} \begin{cases} C_i + C_o & \text{if } \tau_o\text{next to } \tau_i \\ C_i + C_o + \sum_{\tau_h\in\mathcal{I}_{IO,L(\tau_i,\tau_o)}\setminus\{\tau_{i_j},\tau_o\} \mid (\tau_h \text{ after } \tau_i \wedge \tau_h \text{ before } \tau_o)} C_h + \delta_{IO,L(\tau_i,\tau_o)} & \text{otherwise,} \end{cases} \tag{15}$$

and $\delta_{IO,L(\tau_i,\tau_o)}$ account for the idle timethat can occur while scheduling the tasks in between τ_i and τ_o. The constraint $a_oC_o - a_i \leq l_{L(\tau_i,\tau_o)}$ depends on both a_o and a_i. To accomplish the latency constraint it is sufficient to play with the relative position of τ_i and τ_o within $\mathcal{I}_{IO,L(\tau_i,\tau_o)}$.

Intermediate An intermediate independence group $\mathcal{I}_{int,L(\tau_i,\tau_o)}$ to an I/O latency contribute with a constant value $L_{int,L(\tau_i,\tau_o)} = \sum_{\tau_h\in\mathcal{I}_{int,L(\tau_i,\tau_o)}} C_h$.

With Equations (9), (12), (13), (14) and (15) we have decomposed complex latency relations into independence groups. Combining timing conditions from different I/O latencies it would be possible to build an optimization problem acting on intra-group ordering only. For space reasons we could not describe the similar latency conditions for the dependence grouping. Nonetheless, it is worthy to note that they have been derived as inter-group scheduling relations only.

6 Case Studies

We applied the grouping algorithms to several case studies, all of which are derived from avionic applications. Table 1 shows the results. The columns "Tasks" and "Precs" display the number of tasks and precedences, respectively. Column "Dep." shows the number of groups created by the dependent grouping algorithm, while the columns "Indep. F" and "Indep. B" refer to the forward and backward independence grouping algorithms.

Table 1: Case studies grouping results

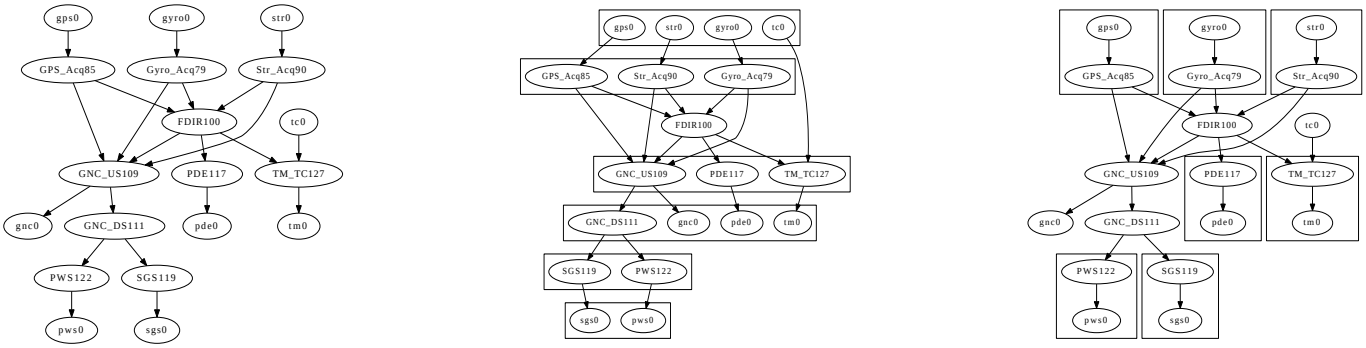
Name	Tasks	Precs.	Dep.	Indep. F	Indep. B
FAS	19	21	12	7	7
FAS_complete	236	329	183	12	12
Asm	14	18	8	7	7
Asm_complete	375	428	368	6	6
CDV_mono	311	49	310	2	2
CDV	511	396	505	3	3
Master	399	305	94	74	74

Table 2: FAS task set configuration

Task	Exec. time	Period	Task	Exec. time	Period
$gps0$	10	1000	TM_TC127	100	1000
$gyro0$	10	1000	$gnc0$	10	1000
$str0$	10	1000	GNC_DS111	300	1000
GPS_Acq85	30	1000	$pde0$	10	1000
$Gyro\_Acq79$	30	1000	$tm0$	10	1000
str_Acq90	30	1000	$PWS122$	30	1000
$FDIR100$			$pws0$	10	1000
$tc0$	10	1000	$SGS119$	30	1000
GNC_US109	210	1000	$SGS0$	30	1000
$PDE117$	30	1000			

For all case studies, the dependent grouping algorithm creates more groups than the independence grouping algorithms. Apparently, task chains are rather short, such that the dependent grouping algorithm can group only few tasks together. In contrast, there are relatively few independence groups. An exception to this is the case study "Master", where the number of dependent and independence groups is relatively balanced. A closer look on the structure of the task graphs revealed that in most cases, the task graph consists of several unconnected components. An effect of this is that tasks of these components end up in the same independence groups, which lowers the number of these groups. Future work could take unconnected components into account to increase the number of independence groups.

With the FAS benchmark we do further investigation of our grouping framework. Figure 3(a) describes the graph representation for the precedence constraints of the FAS task set. Figure 3(b), and 3(c) are respectively the independence group and the dependence group classification for those tasks. The FAS benchmark we analyze is a modified version with mono-rate tasks; the task period has been set to $1000ms$ and the execution times, expressed in milliseconds, have been set to fit into a $1000ms$ period, Table 2. To evaluate the impact of grouping onto I/O latencies, we have defined



(a) Functional graph representation

(b) Independence grouping with the forward algorithm

(c) Dependence grouping

Figure 3: Functional graph and grouping to the FAS benchmark.

two I/O latencies for the FAS benchmark. The first, $L(gps0, gnc0)$ with $gps0$ the input task and $gnc0$ the output task, and the second $L(str0, pws0)$ with $str0$ the input task and $pws0$ the output task. By defining the optimal latency as the smallest latency, we can have an optimization problem where all the possible task scheduling are verified looking for the smallest I/O latency. We consider first the case where single latencies are applied, then we combine the two latencies into the same optimization problem.

Without grouping, the optimal latencies are respectively $275ms$ and $605ms$ for $L(gps0, gnc0)$ and $L(str0, pws0)$. In case of independence grouping (with the forward grouping algorithm) the optimal $L(gps0, gnc0)$ latency become $465ms$, while for the $L(str0, pws0)$ is $845ms$. In case of dependence grouping the optimal $L(gps0, gnc0)$ latency stays $275ms$ (as the no-grouping case), while for the $L(str0, pws0)$ it is $835ms$.

Combining the two latencies, we could look for the maximum between them, $max\{L(gps0, gnc0), L(str0, pws0)\}$. The latencies results into $615ms$ with no grouping applied, $855ms$ with the independence grouping (forward independence grouping) and $835ms$ with the dependence grouping. Results are slightly larger than the single latency case, due to the combination of the two requirements.

As we saw, the grouping reduces the scheduling possibilities. This introduces some pessimism into the task timing

constraints. The pessimism depends on the case study considered, but also on the latencies applied. On the other hand, with the grouping the scheduling complexity is drastically reduced since the number of elements to be scheduled is smaller than the case with no grouping, Table 1.

In future works we will apply latency constraints to better explore Equations (13), (14) and (15) as constraints on the task activations. The relations with the task activations within groups will be explored.

7 Conclusions

In this paper we introduced the notion of grouping to classify tasks with respect to their functional requirements. We proposed two different grouping policies according to the notion of functional dependence and functional independence. The grouping has been applied to the scheduling problem to combine both the timing constraints and the functional classification.

As an initial work, the grouping scheduling approach succeeded in decomposing the task scheduling problem into components (groups). The I/O latencies have been also decomposed into groups allowing to better characterize how they vary within the groups. We have also formalized the potentiality of the task grouping in reducing the complexity of the scheduling problem from the NP-hardness of realistic systems to a component-based problem.

In future stages, we intend to apply the grouping and the decomposed latencies to a structured sensitivity analysis on the activation parameters a. This to provide an useful feedback at system design time and also to compensate part of the pessimism introduced with the grouping. Extensions to this work will also release some of the assumptions made i.e., non preemptability or mono-rate, to generalize the grouping approach.

References

[1] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[2] J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 360–369, March 1990.

[3] E. Grolleau and A. Choquet-Geniet, "Off-line computation of real-time schedules by means of petri nets," in *Workshop On Discrete Event Systems, WODES2000*, ser. Discrete Event Systems: Analysis and Control. Ghent, Belgium: Kluwer Academic Publishers, 2000, pp. 309–316.

[4] C. Ekelin, "An optimization framework for scheduling of embedded real-time systems," Ph.D. dissertation, Chalmers University of Technology, 2004.

[5] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.

[6] M. Spuri and J. Stankovic, "How to integrate precedence constraints and shared resources in real-time scheduling," *Computers, IEEE Transactions on*, vol. 43, no. 12, pp. 1407–1412, 1994.

[7] S. Goddard, "Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application," in *3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 1997, pp. 60–71.

[8] S. Goddard and K. Jeffay, "Managing latency and buffer requirements in processing graph chains," *Comput. J.*, vol. 44, no. 6, pp. 486–503, 2001.

[9] L. Cucu-Grosjean and Y. Sorel, "Non-preemptive scheduling algorithms and schedulability conditions for real-time systems with precedence and latency constraints," INRIA Rocquencourt, Tech. Rep. 5403, December 2004.

[10] L. Cucu, R. Kocik, and Y. Sorel, "Real-time scheduling for systems with precedence, periodicity and latency constraints," in *Proceedings of Real-time and Embedded Systems*, 2002, pp. 26–28.

[11] P. M. Yomsi and Y. Sorel, "Non-schedulability conditions for off-line scheduling of real-time systems subject to precedence and strict periodicity constraints," in *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation, (ETFA)*, 2006.

[12] R. K. Clark, "Scheduling dependent real-time activities," School of Computer Science, Carnegie Mellon University, Tech. Rep., 1990.

[13] M. D. Natale, M. Di, N. John, and J. A. Stankovic, "Dynamic end-to-end guarantees in distributed real time systems," in *Proceedings of RealTime System Symposium, (RTSS)*, 1994.

[14] L. Santinelli, W. Puffitsch, C. Pagetti, and F. Boniol, "Scheduling with functional and non-functional requirements: the sub-functional approach," in *WiP session at 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[15] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, pp. 34–40, March 2005.

[16] S. Baruah and J. Goossens, "Scheduling real-time tasks: Algorithms and complexity," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, 2003.

[17] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, "Deterministic execution model on cots hardware," in *Proceedings of the 25th international conference on Architecture of Computing Systems*, ser. ARCS'12, 2012, pp. 98–110.

[18] M. Lauer, J. Ermont, F. Boniol, and C. Pagetti, "Latency and freshness analysis on ima systems," in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, Sept., pp. 1–8.