Feasibility Analysis of Conditional DAG Tasks is co-NP^{NP}-Hard

(Why This Matters)

Sanjoy Baruah[*]
Department of Computer Science & Engineering
Washington University in Saint Louis
Saint Louis, Missouri, USA
baruah@wustl.edu

ABSTRACT

Feasibility-analysis algorithms have traditionally been required to have running times no worse than pseudo-polynomial in their inputs, in order to be considered *efficient*. But this is changing: motivated by a vast improvement in the performance of Integer Linear Programming (ILP) solvers, some recent work has begun to consider the limited use of ILP solvers as acceptably efficient for the purposes of feasibility analysis. In this paper, a characterization is proposed for the class of feasibility-analysis problems that can be solved efficiently under this more expansive interpretation of efficiency. This characterization is applied to the conditional directed acyclic graph (DAG) workload model, and a demarcation is identified between the feasibility-analysis problems on DAGs that are efficiently solvable using ILP solvers and those that are not.

KEYWORDS

Feasibility analysis; Computational Complexity; Conditional Directed Acyclic Graphs; Integer Linear Programming

ACM Reference Format:
Sanjoy Baruah. 2021. Feasibility Analysis of Conditional DAG Tasks is co-NP^{NP}-Hard: (Why This Matters). In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3453417.3453422

1 INTRODUCTION

Safety-critical systems are required to have the correctness of their run-time behavior verified prior to deployment: such verification is usually done by analysis of models of the run-time behavior. In choosing a model for this purpose, one is driven by two concerns that are typically at odds with each other. On the one hand, the model should be *expressive* in order that it may accurately represent the relevant characteristics of the system being modeled. On the other, it is necessary that we be able to design *efficient* algorithms that let us derive interesting properties of the model, if it to be of much use in system design and analysis.

Thanks to the compounding effects of Moore's Law applied over decades, there has been an enormous increase in computing capabilities over time. As a consequence, our understanding of what forms of analysis may be considered "efficient" has also evolved. In this paper, we consider a particular aspect of run-time correctness – the ability to meet deadlines – that has long been a prime focus of the real-time scheduling community. The pre-run-time analysis of systems to check whether they can be scheduled to always meet their deadlines during run-time is called *feasibility analysis*. In the early years of the discipline of real-time computing, feasibility analysis algorithms were required to have worst-case running times that are low-degree polynomials of input size in order to be considered efficient. Examples of efficient algorithms of this first generation include the utilization-based schedulability tests [10, 16] for Earliest-Deadline First (EDF) and Rate-Monotonic scheduling of collections of independent implicit-deadline sporadic tasks ("Liu and Layland tasks") upon preemptive uniprocessors. By the mid- to late-1980s, computing capabilities had increased enough that feasibility analysis algorithms with *pseudo-polynomial* running times were considered efficient. Early examples of efficient algorithms of this kind include the preemptive uniprocessor EDF feasibility test for bounded-utilization 3-parameter sporadic task systems [8], and Response-Time Analysis for fixed-priority task systems [14]. Ever since, there appears to have been a sort of consensus that pseudo-polynomial running time equates to efficiency, and there has been a continued quest to identify the most general models for which feasibility analysis can be done in pseudo-polynomial time [3, 19, 21] (please see [20] for an excellent survey on this topic).

More recently, the continued increase in computing capabilities, combined with the increasingly complex nature of many feasibility-analysis problems that one encounters whilst seeking to verify the correctness of modern safety-critical cyber-physical systems, has motivated some real-time scheduling theory researchers to take the first tentative steps past the pseudo-polynomial time barrier. Many such investigations seek to transform a feasibility-analysis problem to some other form such as an integer linear program (ILP) or some satisfiability modulo theories (SMT) [2], which can then be solved by a solver of the appropriate kind (i.e., an ILP solver or an SMT-solver, respectively). Although solving an integer linear program, or deciding satisfiability modulo any non-trivial ground theory, is computationally intractable (NP-hard or harder), excellent off-the-shelf solvers exist that, by incorporating a combination of expert techniques, special-purpose heuristics, and highly optimized implementation, are able to handle surprisingly large problem instances in reasonable amounts of time. In this paper, we focus on feasibility-analysis research efforts that seek to go beyond

[*]Funded in part by National Science Foundation Grants CNS-1814739 and CPS-1932530

RTNS'2021, April 7–9, 2021, NANTES, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453422

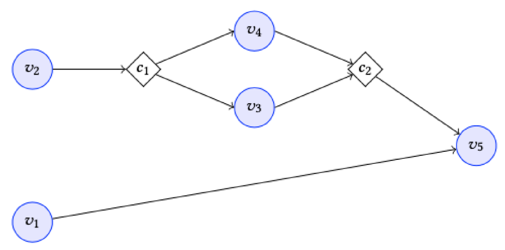

Figure 1: An example Conditional DAG. Conditional nodes are depicted as diamonds, and always occur in matched pairs (in this example, c_1 and c_2). Jobs v_1 and v_2 must both execute. Conditional node c_1 executes after v_2 completes, after which exactly one of the jobs v_3 or v_4 must execute. Conditional node c_2 executes once this job has completed. Job v_5 may only execute after both c_2 and job v_1 have completed.

the pseudo-polynomial time barrier via transformation to integer linear programs (ILPs).

The Conditional DAG Model [6, 18]. In this model, computation is represented as a directed acyclic graph (DAG) in which some vertices, called "conditional nodes", represent boolean expressions that are evaluated during run-time while the other vertices represent non-parallelizable pieces of computation, commonly referred to as "*jobs.*" (The model will be described in detail in Section 2; meanwhile, please see Figure 1 for an example.) The edges leading into and out of conditional nodes may represent control-flow choices that happen based on the outcome of the evaluation of the boolean expressions; the other edges represent precedence constraints between the jobs. The feasibility analysis question for such a conditional DAG asks whether it can be scheduled upon a specified multiprocessor platform, under specified restrictions (e.g., global or partitioned; preemptive or non-preemptive; etc.), to always complete by a specified deadline.

A simpler version of this problem without conditional nodes is equivalent to the widely-studied problem of *makespan-minimization for precedence constrained jobs*: determine whether the jobs can be scheduled to have a duration no larger than the specified deadline upon the provided processors. This problem is known to be NP-hard in the strong sense under most interesting restrictions including global scheduling (when individual jobs are permitted to migrate amongst the processors) [23], and for "typed" systems (where each job is pre-assigned to an individual processor or a specified subset of the processors) [12], regardless of whether preemption is permitted or not. Since conditional DAGs are a generalization of regular DAGs, these hardness results hold for conditional DAGs as well.

ILP solvers. Determining whether an integer linear program (ILP) has a feasible solution was one of the earliest problems shown to be NP-complete [13] (recall that an NP-*complete* problem is both NP-hard and in the class NP). Indeed, it is known to be NP-complete *in the strong sense*[1]; assuming P $\neq$ NP, this implies that ILP solvers with pseudo-polynomial running time cannot be developed. Despite this inherent intractability of ILP, however, the optimization community has recently been devoting immense effort to devise extremely efficient implementations of ILP solvers, and highly optimized libraries with such efficient implementations are widely available today in both open-source and commercial offerings. It is known that the duration taken by an ILP solver to solve a problem tends to correlate very strongly with the size of the problem to be solved — in particular, with the number of variables and constraints in the ILP that is being solved. Modern ILP solvers, particularly when running upon powerful computing clusters, are commonly capable of solving ILPs with tens of thousands of variables and constraints.

[1]A brief review of complexity theory is provided in Section 4 for the reader not familiar with some of this terminology.

This Research, and its Significance. We have seen above that most feasibility-analysis problem for DAGs are NP-hard in the strong sense. Strong NP-hardness rules out the existence of pseudo-polynomial time algorithms for solving these problems (assuming P≠NP). Hence as real-time scheduling theory begins to consider the use of ILP-solvers, it seems appropriate to explore their applicability to the feasibility analysis of real-time systems that are modeled as DAGs — this is the subject of the research reported here. We propose (Definition 1) a classification of such feasibility analysis problems into *ILP-tractable* and *ILP-intractable*. Informally speaking —this will be formalized and made precise in Section 4— we consider a problem to be ILP-tractable if and only if it can be solved by a polynomial-time procedure that may in addition make a few calls to an ILP solver. We provide a separation between these two classes by identifying those modeling features for which feasibility-analysis remains ILP-tractable, and those that render the problem ILP-intractable. Identifying such a demarcation is significant since it provides guidance to system developers as to what features they should seek to avoid in their designs, in order to be able to retain ILP-tractability of analysis.

Organization. The remainder of this paper is organized in the following manner. In Section 2 we provide a more complete description of the DAG-based real-time workload model considered in this work. In Section 3 we briefly summarize our current state of knowledge concerning the representation of DAG feasibility problems as ILPs. In Section 4 we provide a succinct primer on relevant aspects of computational complexity; in Section 5, we interpret currently-known results in the light of these aspects. In Section 6 we present our main technical results, establishing that a commonly-used feature of real-time DAG workload models cannot be efficiently represented as an ILP. In Section 7 we discuss the implications of these technical results to ILP-based schedulability analysis. We conclude in Section 8 by enumerating some possible directions for follow-up research.

2 THE CONDITIONAL DAG MODEL

The models used in scheduling theory for representing real-time workloads that are to be implemented upon multiprocessor platforms should be capable of exposing the parallelism that may exist within these workloads. The *sporadic DAG model* [7] (see [5, Chapter 21] for a text-book description) was proposed for this purpose. A task in this model is specified as a 3-tuple (G, D, T), where G is a directed acyclic graph (DAG), and D and T are positive integers representing the relative deadline and period parameters of the task respectively. The task repeatedly releases *dag-jobs*, each of which is a collection of (sequential) jobs. Successive dag-jobs are released a duration of at least T time units apart. The DAG G is specified as $G = (V, E)$, where V is a set of vertices and E a set of directed edges between these vertices. Each $v \in V$ represents the execution of a sequential piece of code (a "job"), and is characterized by a worst-case execution time (WCET). The edges represent dependencies between the jobs: if $(v_1, v_2) \in E$ then job v_1 must complete execution before job v_2 can begin execution. (Job v_1 is called a *predecessor* job of v_2, and job v_2 is called a *successor* job of v_1.) Jobs that are not predecessors or successors of each other, either directly or transitively, may execute simultaneously upon different processors. A release of a dag-job of the task at time-instant t means that all $|V|$ jobs $v \in V$ are released at time-instant t. If a dag-job is released at time-instant t then all $|V|$ jobs that were released at t must complete execution by time-instant $t + D$.

Conditional nodes. Like a regular sporadic DAG, a conditional DAG [6, 18] is specified as a 3-tuple (G, D, T), where $G = (V, E)$ is a DAG, and D and T are positive integers denoting (as with regular DAGs) the relative deadline and period parameters of the task. *Conditional nodes* are special vertices in V that are defined in matched pairs, that together define a *conditional construct*. Let (c_1, c_2) be such a pair in the DAG $G = (V, E)$ — see Figure 2. Informally speaking, vertex c_1 can be thought of as representing a point in the code where a conditional expression is evaluated and, depending upon the outcome of this evaluation, control will subsequently flow along exactly one of several different possible paths in the code. It is required that all these different paths meet again at a common point in the code, represented by the vertex c_2. More formally,

(1) There are multiple outgoing edges from c_1 in E. Suppose that there are exactly k outgoing edges from c_1 to the vertices $s_1, s_2, \ldots, s_k$, for some $k > 1$. We call k the *branching factor* of this conditional construct. (The branching factor for an "if-then-else" condition is 2.) Then there are exactly k incoming edges into c_2 in E, from the vertices $t_1, t_2, \ldots, t_k$.
(2) For each $\ell \in \{1, 2, \ldots, k\}$, let $V'_\ell \subseteq V$ and $E'_\ell \subseteq E$ denote all the vertices and edges on paths reachable from s_ℓ that do not include vertex c_2. By definition, s_ℓ is the sole source vertex of the DAG $G'_\ell \overset{\text{def}}{=} (V'_\ell, E'_\ell)$. It must hold that t_ℓ is the sole sink vertex of G'_ℓ.
(3) It must hold that $V'_\ell \bigcap V'_j = \emptyset$ for all $\ell, j, \ell \neq j$. Additionally, with the exception of (c_1, s_ℓ) there should be no edges in E into vertices in V'_ℓ from vertices not in V'_ℓ, for each $\ell \in \{1, 2, \ldots, k\}$. I.e., $E \bigcap ((V \setminus V'_\ell) \times V'_\ell) = \{(c_1, s_\ell)\}$ should hold for all ℓ.

Edges (v_1, v_2) between pairs of vertices neither of which are conditional vertices represent precedence constraints exactly as in traditional sporadic DAGs, while edges involving conditional vertices represent conditional execution of code. More specifically, let (c_1, c_2) denote a defined pair of conditional vertices that together define a conditional construct. The semantics of conditional DAG execution mandate that

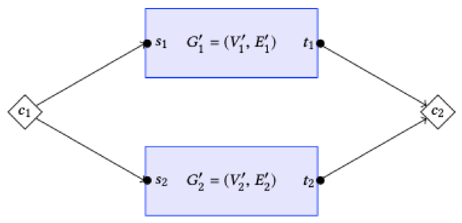

Figure 2: A canonical conditional construct with branching factor 2. Vertices s_1 and t_1 (vertices s_2 and t_2, resp.) are the sole source vertex and sink vertex of G'_1 (G'_2, resp.).

- After the job c_1 completes execution, exactly one of its successor jobs becomes eligible to execute; it is not known beforehand which successor job may execute.
- Job c_2 begins to execute upon the completion of exactly one of its predecessor jobs.

It is important to note that the conditional expressions may evaluate differently during different executions of a conditional DAG. Let $\mathbb{J}$ denote all possible complete collections of jobs that comprise a single dag-job of the conditional DAG, along with the precedence constraints amongst these jobs that are imposed by the edges of the DAG. Thus each $J \in \mathbb{J}$ denotes a collection of precedence-constrained jobs obtained by completely executing through the DAG once, taking into account the conditional branches within it. There may in general be exponentially many different flows through a graph: consider for example the following skeleton of code (here each `(Ci)` represents a boolean condition that may evaluate to either true or false, and each `{Sij}` a block of straight-line code):

```
if (C1) then {S11} else {S12}
if (C2) then {S21} else {S22}
. . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . .
if (Cn) then {Sn1} else {Sn2}
```

If the different `(Ci)`'s are independent, then this code fragment may experience any of 2^n different execution flows through it depending upon whether the `(Ci)`'s evaluate to true or false; hence $|\mathbb{J}|$, the number of precedence-constrained collections of jobs in $\mathbb{J}$, may be exponential in the number of vertices in G. As a consequence, algorithms for the analysis of conditional DAGs that are based upon explicitly examining each $J \in \mathbb{J}$ will necessarily have exponential worst-case running time.

Alternative nodes [25]. The **C-DAG model** was recently proposed by Zahaf et al. [25] as a further generalization to the conditional DAG model. One of the additional features introduced in this model is that of *alternative nodes*. Alternative nodes are syntactically essentially identical to conditional nodes: like conditional nodes, they occur in matched pairs and may be nested. However they are interpreted very differently: they model alternative implementation possibilities for parts of the task. Prior to run-time the system designer may choose any one of the alternative implementations that are available between a matched pair of implementation nodes.

Summarizing Complexity Results. We now briefly summarize relevant current knowledge regarding the computational complexity of feasibility-analysis problems for DAGs. As mentioned earlier, most multiprocessor feasibility-analysis problems for DAGs are known to be NP-hard in the strong sense. The first such result, to our knowledge, dates back to 1975 when Ullman showed [23] that it is NP-hard in the strong sense to determine whether a given DAG can be scheduled to meet a specified deadline under global scheduling upon an identical multiprocessor platform, under both preemptive and non-preemptive scheduling. (An even earlier result [13], showing that non-preemptive partitioned scheduling of independent jobs is NP-hard in the strong sense, of course implies that non-preemptive partitioned scheduling of DAGs is also NP-hard in the strong sense; however [13] did not explicitly consider DAGs). Jansen subsequently showed [12] that feasibility analysis of DAGs is NP-hard in the strong sense if each job is pre-assigned to a particular processor, again under both preemptive and non-preemptive scheduling. Since these basic problems are already NP-hard in the strong sense, so are the feasibility-analysis problems for the more general models that allow for enhanced features such as heterogeneous processor types, conditional nodes, alternative nodes, etc. A recent work [17] studied the list-scheduling [11] of conditional DAGs for a given fixed list-ordering, and showed that determining the minimum schedule duration here is also an NP-hard problem.

3 ILP REPRESENTATIONS OF DAG-SCHEDULABILITY

We now briefly review the current state of the art regarding the representation of feasibility-analysis problems for DAGs as integer linear programs. We start with a definition:

Definition 1 (ILP-tractability). *A schedulability-analysis problem is said to be* ILP-tractable *if it can be solved by an algorithm that has polynomial running time and may in addition make polynomially many calls to an ILP-solver.*

Problems that are not ILP-tractable are ILP-intractable. □

This definition captures the intuitive notion that with the availability of fast modern ILP solvers, it may reasonably be considered quite efficient to make a few calls to such a solver. Hence, we have chosen to require that an efficient feasibility-analysis algorithm run in polynomial time and make polynomially many calls to such a solver.

Standard techniques from the "traditional" (i.e., Operations Research (OR)/ Theoretical Computer Science) scheduling-theory literature, such as *sequence-position decision variables* and *precedence decision variables* (see, e.g, [1, Appendix C] for a text-book introduction to these and similar techniques) can be applied to obtain ILP representations of most forms of non-preemptive feasibility analysis for "regular" DAGs (i.e., those without conditional nodes) in polynomial time; hence, the corresponding feasibility-analysis problems are all ILP-tractable. When preemption is permitted, these techniques are not by themselves adequate for representing feasibility as an ILP, and some other techniques that can be used (in particular, *time-indexed decision variables* [1, page 473]) tend to yield ILPs of size exponential (or at least, pseudo-polynomial) in the representation of the DAG for which feasibility-analysis is sought. A paper [4] that was presented at RTNS last year integrated some traditional OR techniques with some other ideas (such as the *demand-bound function* [8]) that are explicitly from the domain of real-time scheduling theory to obtain, in polynomial time, a polynomial-sized ILP for representing the feasibility-analysis problem for DAGs in which each job is pre-assigned to a particular processor (i.e., the problem shown to be NP-hard in the strong sense in [12]).[2] Hence the results in [4, 9] bear witness to the fact that this feasibility-analysis problem is also ILP-tractable. It is relatively straightforward to extend the method of [4, 9] to handle several generalizations including partitioned scheduling when jobs are not pre-assigned to individual processors, fixed-job-priority scheduling, and choosing optimally between a pair of matched alternative nodes [25]; hence, feasibility analysis for DAG-based real-time workload models that incorporate some or all of these features can be represented as ILPs in polynomial time, and are all ILP-tractable.

The main technical result that we will derive in this paper is that *certain feasibility-analysis problems for DAG models that allow for conditional nodes are ILP-$\underline{\text{in}}$tractable* — they cannot be solved in polynomial time with polynomially many calls to an ILP-solver (modulo certain assumptions, such as P ≠ NP, that are very widely believed to be true in the theoretical computer science community). Furthermore, we will see that ILP-intractability holds even if the conditional constructs that are present in our DAGs are very simple: not nested, and with just a single job along each branch of the conditional construct. It thus appears that the precise demarcation between ILP-tractability and ILP-intractability is with conditional nodes: their presence, even in very modest form, introduces ILP-intractability.

4 COMPUTATIONAL COMPLEXITY: SOME BACKGROUND

In this section we provide a brief introduction to those concepts of computational complexity theory that are needed in the remainder of this manuscript. (In order to keep things simple the presentation in this section is intentionally informal and not always precise: for instance, while most of the concepts discussed below differ in their applicability to *decision problems* – those for which there is a "yes/ no" answer – and *optimization problems*, we do not make this distinction here but treat both decision and optimization problems in similar fashion.)

The class $\underline{\text{P}}$ of problems that are known to be *solved* by algorithms with running time polynomial in the size of their inputs, and the class $\underline{\text{NP}}$ of problems for which claimed solutions can be *verified* by algorithms with running time polynomial in the size of their inputs, are foundational cornerstones of computational complexity theory. It is very widely believed that P $\subsetneq$ NP; i.e., there are polynomial-time verifiable problems that cannot be solved in polynomial time.

The class NP may be further sub-divided into the sub-classes *NP in the strong sense* and *NP in the weak sense*. Somewhat informally, a problem is in the class NP in the weak sense if its specification includes numbers, and there is an algorithm for solving the problem that has running-time polynomial in the *values* of the numbers of

[2]A subtle error in the ILP formulation in [4] was identified, and a fix provided, by Ben-Amor [9, Appendix A].

the input instance – i.e., there is a *pseudo*-polynomial-time algorithm for solving it. Otherwise, it is in the class NP in the strong sense. It is widely believed that the class of problems that are NP in the weak sense is strictly contained in the class of NP problems: i.e., the consensus (although unproved) relationship amongst the three classes of problems is as shown in the Venn diagram (left) in Figure 3. As mentioned earlier, determining whether an ILP has a feasible solution is known to be NP-complete in the strong sense [13]; its assumed position in the complexity hierarchy is depicted in the Venn diagram: in NP, but not in the weak sense.

Recall our discussion in Section 1, that in the context of schedulability analysis "*efficiency*" had initially meant polynomial-time; subsequently pseudo-polynomial time; and is now beginning to be considered to mean "solvable by an ILP" as computational capabilities continue to increase over time. Notice the direct co-relation between this evolving understanding of what efficiency means, and the complexity classes discussed above: initially, problems in the class P were considered efficient; later, problems that are in NP in the weak sense. The recent inclination in real-time scheduling theory to equate efficiency with ILP-tractability is essentially asserting that thanks to the combination of improvements in ILP-solvers and the compounding effects of Moore's law, we may consider that *problems representable as ILPs are efficiently solvable* despite being NP-hard in the strong sense.

The *polynomial-time hierarchy* [22] extends computational complexity theory beyond the classes P and NP by considering computers equipped with an *oracle*: a "black box" that is able to solve a specific decision problem in a single step, and hence in $\Theta(1)$ (i.e., constant) time. The complexity class P^{NP} denotes the class of all problems that can be solved in polynomial time by a computer that is equipped with an oracle that solves some NP-complete problem. In a similar vein, the complexity class NP^{NP} denotes the class of all problems that can be *verified* in polynomial time by a computer that is equipped with an oracle that solves some NP-complete problem. (And just as co-NP denotes the class of problems whose complement problems are in NP, co-NP^{NP} denotes the class of problems whose complement problems are in NP^{NP}.)

The relationship amongst the six complexity classes we have discussed above (P, NP, co-NP, P^{NP}, NP^{NP}, and co-NP^{NP}) is depicted in Figure 3 on the right; the arrows represent the SUBSET-OF relationship $\subseteq$. It is widely assumed (but unproven) that all of these SUBSET-OF relationships are strict (i.e., $\subsetneq$).

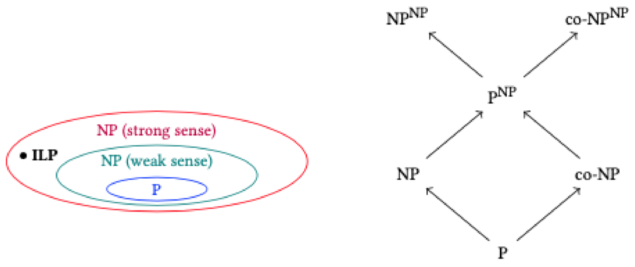

Figure 3: Relationship between complexity classes. The innermost (blue) circle represents the problems in P, the intermediate (teal) one includes problems that are in NP in the weak sense, and the outermost (red) circle further includes problems that are in NP in the strong sense.
The lattice on the right depicts some additional complexity classes.

5 IMPLICATIONS FOR CONDITIONAL DAG FEASIBILITY-ANALYSIS

The relationship between the complexity classes discussed in Section 4 above, and the fact that ILP is complete for the class NP, has immediate implications for feasibility analysis of DAGs.

LEMMA 1. *ILP-tractability is equivalent to membership in the complexity class* P^{NP}.

Proof. As discussed in Section 4 above, P^{NP} is the complexity class of all problems that can be solved in polynomial time by a computer that is equipped with an oracle for solving some NP-complete problem. Since solving ILPs is an NP-complete problem [13], an ILP-solver could be such an oracle. Any problem solved in polynomial time by a computer equipped with an ILP-solver as an oracle is therefore, by definition of the complexity class, in P^{NP}. □

Recall from complexity theory that a problem is said to be *complete* for a complexity class of the kinds discussed in Section 4 above if (i) it belongs to the class; and (ii) it is *hard* for the class, i.e., any problem in the class can be reduced to this problem in polynomial time. Our major result in the remainder of this paper is a proof, in Section 6, that feasibility analysis for conditional DAGs is hard for the complexity class co-NP^{NP}. Since it is widely believed that P^{NP} is not equal to co-NP^{NP}, this implies that it is highly unlikely that feasibility analysis for conditional DAGs is ILP-tractable.

6 AN ILP-INTRACTABLE SCHEDULING PROBLEM

In this section, we will show that the following feasibility-analysis problem on conditional DAGs:
Determine whether a given conditional DAG in which each job is pre-assigned to a particular processor is guaranteed to always complete by a specified deadline
is hard for the complexity class co-NP^{NP}; based on the results discussed in Section 5 above, this would imply that this problem is ILP-intractable. We show this hardness by presenting a polynomial-time reduction to this problem from the $\forall\exists$3SAT problem, which is defined in the following manner:

DEFINITION 2 (THE $\forall\exists$3SAT PROBLEM).
INSTANCE. *A Boolean formula $\phi(\vec{x},\vec{y})$ in 3CNF (Conjunctive Normal Form – i.e., as the "and" of clauses each comprising exactly 3 literals)*
QUESTION. *Is is true that $(\forall\vec{x})(\exists\vec{y})\phi(\vec{x},\vec{y})$?* □

It is known [22, 24] that the $\forall\exists$3SAT problem is complete for the complexity class co-NP^{NP}.

THEOREM 1. *It is* co-NP^{NP}*-hard to determine whether a given conditional DAG in which each job is restricted to execute upon a specified processor, is guaranteed to always complete execution by a specified deadline.*

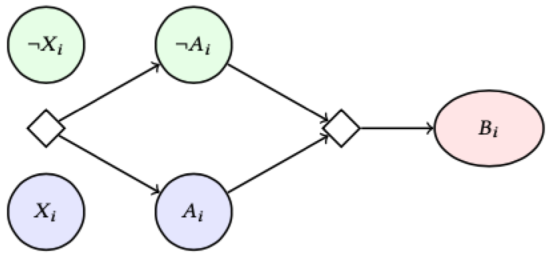

Figure 4: The jobs constructed for boolean variable x_i.

Proof. We show this by reducing a given $\forall\exists$3SAT expression with $(n_x + n_y)$ boolean variables and m 3CNF clauses

$$\forall\,(x_1, x_2, \ldots, x_{n_x})\,\exists\left(y_1, y_2, \ldots, y_{n_y}\right)\bigwedge_{i=1}^{m}\left(\ell_{k,1}\vee\ell_{k,2}\vee\ell_{k,3}\right) \quad (1)$$

where each $\ell_{k,j}$ is one of the x_i or y_j boolean variables or its negation, to a conditional DAG with

- $(7n_x + 2n_y + 3m)$ nodes, of which $2n_x$ are conditional nodes and the rest represent jobs;
- $(5n_x + 3m)$ edges;
- $(3n_x + n_y + m)$ processors; and
- deadline $D = 4$

that is feasible if and only if the $\forall\exists$3SAT expression evaluates to true. The reduction proceeds in the following manner.

FOR EACH BOOLEAN VARIABLE x_i. We define four jobs labeled X_i, $\neg X_i$, A_i, $\neg A_i$ with unit execution requirements, and a single job B_i with execution requirement 3. (We will say that the job X_i *corresponds to* the literal x_i, and the job $\neg X_i$ *corresponds to* the literal $\neg x_i$.)

The edges connecting these vertices are as shown in Figure 4: we have a conditional construct (start-node and associated end-node), with A_i on one branch and $\neg A_i$ on the other, and an edge from the end-node of the conditional construct to the node B_i. Job B_i is assigned to processor $P_{1,i}$. Jobs X_i and A_i are both assigned to processor $P_{2,i}$. Jobs $\neg X_i$ and $\neg A_i$ are both assigned to processor $P_{3,i}$. (Jobs that are assigned to the same processor are shaded with the same color in the diagram above.)

Since the deadline is at time-instant 4 and job B_i has an execution duration of 3, B_i must begin executing no later than time-instant 1 if it is to complete by the deadline. Hence, the conditional construct must complete execution no later than time-instant 1, implying that exactly one of the jobs $\{A_i, \neg A_i\}$ must execute during the time-interval $[0, 1]$. Since job A_i (job $\neg A_i$, respectively) is assigned to the same processor as job X_i (job $\neg X_i$, resp.), this in turn implies that

FACT 1. *For each i, $1 \le i \le n_x$, at most one of the jobs $\{X_i, \neg X_i\}$ completes execution by time-instant 1 in any schedule in which the deadline is met.*

FOR EACH BOOLEAN VARIABLE y_j. We define two jobs labeled Y_j and $\neg Y_j$ with unit execution requirements, both assigned to the same processor $P_{4,j}$. Analogously to above, we will say that the job Y_j (job $\neg Y_j$, respectively) *corresponds to* the literal y_j (the literal $\neg y_j$, resp.).

Since both jobs Y_j and $\neg Y_j$ are assigned to the same processor, it follows that

FACT 2. *For each j, $1 \le j \le n_y$, at most one of the jobs $\{Y_j, \neg Y_j\}$ completes execution by time-instant 1 in any schedule in which the deadline is met.*

Hence by time-instant 1 in any schedule in which the deadline is met, at most one of each pair of jobs $\{X_i, \neg X_i\}$, and at most one of each pair of jobs $\{Y_j, \neg Y_j\}$, could have completed execution. The literals to which the executed jobs correspond can be considered to comprise a truth assignment to the boolean variables $\left(\{x_i, x_2, \ldots, x_{n_x}\} \bigcup \{y_1, y_2, \ldots, y_{n_y}\}\right)$; this leads to the conclusion

FACT 3. *The jobs that have completed execution by time-instant 1 in any schedule in which the deadline is met are those corresponding to the literals in some (complete or incomplete) truth-assignment to the boolean variables of Expression 1.*

FOR EACH CLAUSE $(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3})$. We will define three unit-sized jobs $C_{k,1}, C_{k,2}$, and $C_{k,3}$, all of which are assigned to the same processor $P_{5,k}$, and show that at least one of these jobs will be eligible to execute at time-instant 1 if and only if the truth-assignment of Fact 3 above causes the clause $(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3})$ to evaluate to true; i.e., at least one of the three literals $\ell_{k,1}, \ell_{k,2}$, or $\ell_{k,3}$, is assigned the truth value $\boldsymbol{T}$ (for "TRUE"). We do so by having a single incoming edge into job $C_{k,1}$ from the job corresponding to the literal $\ell_{k,1}$, a single incoming edge into job $C_{k,2}$ from the job corresponding to the literal $\ell_{k,2}$, and a single incoming edge into job $C_{k,3}$ from the job corresponding to the literal $\ell_{k,3}$.
We point out that some boolean variable not being assigned a truth value (i.e., the truth assignment of Fact 3 not being a complete one) cannot cause some job to become eligible to execute, that would subsequently be rendered ineligible if the truth assignment were completed. That is,

FACT 4. *The number of $C_{k,\ell}$ jobs that become eligible to execute at time-instant 1 is maximized when the truth assignment of Fact 3 is a complete one.*

That concludes our description of the construction of our conditional DAG from Expression 1. We now prove that it can be scheduled to always complete by its deadline of 4 if and only if Expression 1 is valid.

LEMMA 1.1. *If Expression 1 is true, then the conditional DAG constructed above can be scheduled to always complete by its deadline.*

Proof. Suppose that Expression 1 is valid: for any assignment of truth values to the boolean variables $(x_1, x_2, \ldots, x_{n_x})$, there is an assignment of truth values to the boolean variables $\left(y_1, y_2, \ldots, y_{n_y}\right)$ that causes each of the m clauses of Expression 1 to evaluate to true. We point out that

(1) Each assignment of truth values to the boolean variables
$$(x_1, x_2, \ldots, x_{n_x})$$
can be emulated by executing the appropriate branch of the conditional construct that appears in our conditional DAG. For instance, suppose $x_i \leftarrow \boldsymbol{T}$; the execution of the

conditional construct that causes the job $\neg A_i$ to execute would prevent job $\neg X_i$ from executing, but would permit job X_i to execute, by time-instant 1.

(2) The assignment of truth values to the boolean variables
$$\left(y_1, y_2, \ldots, y_{n_y}\right)$$
that causes each of the m clauses of Expression 1 to evaluate to true can be emulated by executing the appropriate one of the two jobs that were generated for each y_j. Suppose, for instance, that $y_j \leftarrow \boldsymbol{F}$ in this assignment; this can be emulated by executing the job $\neg Y_j$ by time-instant 1.

(3) Hence, each truth-assignment to the boolean variables that cause the m clauses of Expression 1 to evaluate to true can be emulated such that the jobs corresponding to the literals that are true in such a truth-assignment are executed by time-instant 1.

(4) Consequently, at least one of the three jobs $C_{k,1}, C_{k,2}$, and $C_{k,3}$ corresponding to each clause is eligible to execute by time-instant 1, thereby allowing all three jobs to complete execution on their shared processor by the deadline at time-instant 4.

And this concludes the proof of Lemma 1.1. □

Lemma 1.2. *If the conditional DAG constructed above can be scheduled to always complete by its deadline, then Expression 1 is true.*

Proof. Suppose that the conditional DAG we have constructed can be scheduled to always complete by its deadline.

(1) The job B_i that was defined for each variable x_i has execution requirement 3, and so must begin execution no later than time-instant 1 in order to complete by the deadline. Hence the conditional construct defined for the variable x_i must complete execution by time-instant 1.

(2) Since each of the n_x conditional constructs are independent of each other, the choice of which branches of each to execute, which in turn restricts which of the pair of jobs $X_i, \neg X_i$ may execute over the interval $[0, 1]$ for each i, $1 \leq i \leq n_x$, can force each of the 2^{n_x} possible truth-assignments to the variables $\left(x_1, x_2, \ldots, x_{n_x}\right)$ to be emulated by time-instant 1.

(3) For each of these truth-assignments, it must be the case that at least one of the three jobs $C_{k,1}, C_{k,2}$, and $C_{k,3}$ corresponding to the k'th clause is eligible to execute by time-instant 1 (in order that all three of these jobs may complete on their common processor by time-instant 4), for each k, $1 \leq k \leq m$.

(4) Hence it must be possible to execute exactly one of the two jobs $Y_j, \neg Y_j$ upon their shared processor during the time-interval $[0, 1]$ for each j, $1 \leq j \leq n_y$, such that the assignment of truth values to the boolean variables $\left(y_1, y_2, \ldots, y_{n_y}\right)$, when combined with the choice of assignment to the boolean variables $\left(x_1, x_2, \ldots, x_{n_x}\right)$ implied by the conditional branches that were executed, causes each of the clauses to be satisfied.

This establishes that Expression 1 is true, and concludes the proof of Lemma 1.2. □

Taken together, Lemmas 1.1 and 1.2 lead us to conclude that determining whether a conditional DAG can be scheduled to always meet its deadline is co-NP$^{\text{NP}}$-hard. We have thus established the truth of Theorem 1. □

The following corollary is an immediate consequence of Theorem 1 and the widely-held belief that each of the SUBSET-OF relationships depicted in Figure 3 is strict (i.e., $\subsetneq$).

Corollary 1. *There cannot be a polynomial-time procedure that may in addition call an ILP solver polynomially many times, for solving the feasibility-analysis problem for conditional DAGs in which jobs are pre-assigned to processors.* □

Our next corollary follows by observing that all the numerical parameters (i.e., the job execution durations and the overall deadline) characterizing the conditional DAG that was constructed during the reduction in the proof of Theorem 1 are of *value* no larger than polynomial in the size of the input $\forall\exists$3SAT instance (Expression 1); hence, a pseudo-polynomial algorithm for determining its feasibility on the oracle-equipped computer would imply a polynomial-time algorithm for the $\forall\exists$3SAT Problem upon this same computer, thereby showing that P$^{\text{NP}}$ $\equiv$ co-NP$^{\text{NP}}$. Since it is widely believed that this is not the case, we conclude

Corollary 2. *There cannot be a pseudo-polynomial-time procedure that may in addition call an ILP solver pseudo-polynomially many times, for solving the feasibility-analysis problem for conditional DAGs in which jobs are pre-assigned to processors.*

7 DISCUSSION

To our knowledge, most current efforts at representing feasibility analysis problems as ILPs are *ad hoc*. The technical results of Sections 5 and 6 suggest a methodical approach to determining whether a feasibility-analysis (or indeed, any) problem can be solved efficiently by using ILP solvers:

- To determine whether we can efficiently represent a problem as an ILP, we should try to show that the problem is in the complexity class NP.
- To determine whether we can efficiently solve a problem in polynomial time with one or more additional calls to an ILP solver, we should try to show that the problem is in the complexity class P$^{\text{NP}}$.
- If we are not able to make progress in efforts at efficiently solving a problem with the help of an ILP solver, we should check whether the problem is in fact hard for NP$^{\text{NP}}$ or co-NP$^{\text{NP}}$ (in which case our efforts are unlikely to bear fruit).

8 CONTEXT AND FUTURE WORK

Real-time scheduling theory has begun considering the use of ILP solvers to obtain efficient algorithms for solving feasibility analysis problems. In this paper, we report on our findings from a recently-initiated methodical study of what efficiency means from this perspective: how does one recognize when a feasibility-analysis problem can be efficiently solved via ILPs? We have identified the boundaries of what can reasonably be considered to be efficiently solvable – see Figure 5, and have demonstrated how one shows a problem to *not* be efficiently solvable in this framework. Based on this categorization, an important question that merits further investigation is this: what is the most *general* real-time workload model for which the feasibility-analysis problem remains tractable

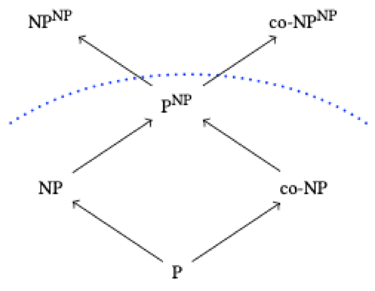

Figure 5: Complexity classes for which efficient ILP-based solutions exist.

(i.e., below the dotted blue line in Figure 5)? We note that this is essentially equivalent to identifying interesting real-time scheduling problems that fall within the complexity class P^{NP} (see [15]).

REFERENCES

[1] Kenneth R. Baker and Dan Trietsch. 2009. *Principles of Sequencing and Scheduling.* Wiley Publishing.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885. https://doi.org/10.3233/978-1-58603-929-5-825

[3] Sanjoy Baruah. 1998. A general model for recurring real-time tasks. In *Proceedings of the Real-Time Systems Symposium.* IEEE Computer Society Press, Madrid, Spain, 114–122.

[4] Sanjoy Baruah. 2020. Scheduling DAGs When Processor Assignments Are Specified. In *Proceedings of the Twenty-Fifth International Conference on Real-Time and Network Systems (RTNS '20).* ACM, New York, NY, USA.

[5] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. 2015. *Multiprocessor Scheduling for Real-Time Systems.* Springer Publishing Company, Incorporated.

[6] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. 2015. The Global EDF Scheduling of Systems of Conditional Sporadic DAG tasks. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems (ECRTS '15).* IEEE Computer Society Press, Lund (Sweden), 222–231.

[7] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. 2012. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 2012).* San Juan, Puerto Rico, 63–72.

[8] S. Baruah, A. Mok, and L. Rosier. 1990. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the 11th Real-Time Systems Symposium.* IEEE Computer Society Press, Orlando, Florida, 182–190.

[9] Slim Ben-Amor. 2021. *Multicore Scheduling of Dependent Tasks with Probabilistic Execution Times.* Ph.D. Dissertation. Sorbonne Université.

[10] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. 2003. Rate Monotonic Scheduling: The Hyperbolic Bound. *IEEE Trans. Comput.* 52, 7 (2003), 933–942.

[11] R. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* 45 (1966), 1563–1581.

[12] Klaus Jansen. 1994. Analysis of scheduling problems with typed task systems. *Discrete Applied Mathematics* 52, 3 (1994), 223 – 232.

[13] R. Karp. 1972. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, New York, 85–103.

[14] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. 1993. *A Practitioner's Handbook for Real-time Analysis.* Kluwer Academic Publishers, Norwell, MA, USA.

[15] Mark W. Krentel. 1988. The complexity of optimization problems. *J. Comput. System Sci.* 36, 3 (1988), 490 – 509.

[16] C. Liu and J. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61.

[17] Alberto Marchetti-Spaccamela, Nicole Megow, Jens Schlöter, Martin Skutella, and Leen Stougie. 2020. On the Complexity of Conditional DAG Scheduling in Multiprocessor Systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[18] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. 2015. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems (ECRTS '15).* IEEE Computer Society Press, Lund (Sweden), 222–231.

[19] Aloysius K. Mok and Deji Chen. 1996. A multiframe model for real-time tasks. In *Proceedings of the 17th Real-Time Systems Symposium.* IEEE Computer Society Press, Washington, DC.

[20] Martin Stigge. 2014. *Real-Time Workload Models: Expressiveness vs. Analysis Efficiency.* Ph.D. Dissertation. Ph.D. thesis, Uppsala University.

[21] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. 2011. The Digraph Real-Time Task Model. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS).* IEEE Computer Society Press, Chicago, 71–80.

[22] L. Stockmeyer. 1976. The Polynomial-Time Hierarchy. *Theoretical Computer Science* 3 (1976), 1–22.

[23] J. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384 – 393.

[24] C. Wrathall. 1976. Complete Sets and the Polynomial-Time Hierarchy. *Theoretical Computer Science* 3 (1976), 23–33.

[25] Houssam-Eddine Zahaf, Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Giuseppe Lipari. 2019. A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: Preemption matters. arXiv:1901.02450 [cs.OS] arXiv.