Modeling the Resource-Constrained Project Scheduling Problem with Answer Set Programming

Mohammed M. S. El-Kholany[1], Martin Gebser[2], Dorde Markovic[2] and Gianluca Zavan[2]

[1]Faculty of Computers and Artificial Intelligence, Cairo University, Egypt
[2]Department of Artificial Intelligence and Cybersecurity, University of Klagenfurt, Austria

Abstract

The Resource-Constrained Project Scheduling Problem (RCPSP) is a long-standing challenge problem in the area of constraint optimization. Its goal is to schedule partially ordered activities, each requiring specific resources, as effectively as possible while respecting the resources' capacity limits. Previous approaches have utilized Constraint Programming or local search techniques for solving the RCPSP to exact or approximate optimality, respectively. While Answer Set Programming (ASP) along with its extensions by difference logic or finite domain constraints has been successfully applied to closely related scheduling problems, these declarative problem solving approaches have so far not been explored for the RCPSP.

In this work, we elaborate on modeling approaches for encoding the RCPSP in plain ASP as well as its extensions by difference logic and finite domain constraints. Interestingly, formulating resource capacity limits is simpler in plain ASP due to the explicit representation of time points, while the use of difference logic or finite domain constraints incurs a certain modeling effort. Our empirical comparison on small- to medium-scale RCPSP instances shows that the simplicity of modeling promotes plain ASP and more than outweighs its larger problem representation size, which counteracts the performance trends typically observed on related scheduling problems.

Keywords

Resource-Constrained Project Scheduling Problem, Answer Set Programming, Modeling and Solving

CEUR-WS.org/Vol-4311/ASPOCP_regular_01.pdf

1. Introduction

The Resource-Constrained Project Scheduling Problem (RCPSP) [1] is a special class of scheduling problems, particularly interesting due to its various applications. Some of the famous use cases include experiments scheduling on the Rosetta/Philae mission [2], scheduling the Airbus A330 assembly line [3], scheduling hazardous material examinations [4], and many others. The idea is to find a schedule of the set of activities within a project while considering limited resources. Each activity takes a certain amount of time and resources. Additionally, activities are subject to precedence constraints, meaning that some activities must finish before others can start. The objective is to minimize the overall project duration while adhering to the resource and precedence constraints. There are several variants of the RCPSP, including cases with stochastic activity durations, renewable and non-renewable resources, and time-dependent resource availability. In this paper, we consider the deterministic RCPSP, where activity durations are known in advance and resources are renewable (i.e., not permanently consumed by a process) with fixed availability. Furthermore, the problem parameters are instance-dependent, i.e., different instances may involve varying numbers of activities, resources, and precedence relations.

The RCPSP was introduced, formalized, and studied as part of operations research dating back to the 1960s (see early surveys [5, 6]), and also serves as the backbone of planning and execution in project management [7, 8, 9]. One can view the RCPSP as an evolution of the project scheduling problem without resource constraints, which was efficiently solved with the Critical Path Method (CPM) [10]. Other tools, such as the Program Evaluation and Review Technique (PERT), were focused on analyzing

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.
m.saadelden@fci-cu.edu.eg (M. M. S. El-Kholany); martin.gebser@aau.at (M. Gebser); dorde.markovic@aau.at (D. Markovic); gianluca.zavan@aau.at (G. Zavan)
0000-0002-1088-2081 (M. M. S. El-Kholany); 0000-0002-8010-4752 (M. Gebser); 0000-0002-1932-975X (D. Markovic); 0009-0003-9748-4338 (G. Zavan)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

scheduling problems with nondeterministic activity durations. However, these tools analyze project networks solely based on the precedence constraints [10, 11] and do not explicitly account for resource limitations, thus motivating the need for more sophisticated formulations [12, 13]. The complexity of RCPSP was established as NP-hard [12] when resources are not fixed.

In the early days, the main approaches to RCPSP solving were Integer Programming [14] and Branch-and-Bound methods [15]. However, due to the problem's complexity, these methods could only solve small instances in manageable time. This led to continuing the research in the direction of heuristics (see [16] for an overview), aiming to provide good-enough solutions without guaranteeing optimality. Moreover, these methods were improved with metaheuristic approaches, such as genetic algorithms, simulated annealing, and tabu search (see [17] for a survey), which were capable of escaping local optima and handling larger instances, but still without providing optimality guarantees. With the beginning of this millennium and advancements of exact solving methods, the RCPSP received renewed interest from Constraint Programming (CP) [18] and Mixed Integer Linear Programming (MILP) [19] communities. Furthermore, these approaches were combined with metaheuristics, such as Large Neighborhood Search (LNS) [3, 20], resulting in significant improvements in solution quality. However, these hybrid methodologies often require expert knowledge of problem-solving tools and problem specifics for successful implementation. Recent work on the RCPSP has explored machine-learning-guided heuristics and reinforcement learning methods to address uncertainty and dynamic scheduling environments [21, 22]. All these efforts resulted in tools capable of solving large industry-scale instances.

So far, CP and MILP are the main exact solving approaches for the RCPSP. Our work explores Answer Set Programming (ASP) [23] as a high-level declarative programming language, which has been successfully applied to related scheduling problems (see, e.g., [24]), but not yet to the RCPSP. As a language oriented primarily towards NP-hard search problems, ASP is naturally suited for this task. Aside from modeling the RCPSP in plain ASP, we will investigate two ASP extensions: Difference Logic (DL) and finite domain constraints. DL aims to improve the modeling and solving of problems with numerical constraints by incorporating a restricted form of linear arithmetics in ASP modulo DL [25]. Similarly, Constraint Answer Set Programming (CASP) [26] integrates finite domain constraints over bounded integers. CASP solving approaches, including specializations to ASP modulo DL, can be grouped into three categories: *hybrid* systems that combine an ASP solver with a CP or MILP solver encompass *acsolver* [27], *ascass* [28], *caspscript* [29], *clingcon* v1 [30] and v2 [31], *clingo*[LP] [25], *dlvhex*[CP] [32], and *ezcsp* [33]; *translations* mapping CASP to MILP or Satisfiability Modulo Theories (SMT) [34] formulations are performed by *aspmt2smt* [35], *dingo* [36], *ezsmt* [37], and *mingo* [38]; the systems *adsolver* [39], *clingcon* v3 [40], *clingo*[DL] [25], *idp* [41], and *inca* [42] implement *extensions* of ASP systems for constraint propagation on theory atoms. Among the latter, *clingcon*, *clingo*[DL], and *inca* augment the ASP system *clingo* [43] by performing specific forms of lazy clause generation [44], a technique from satisfiability solving driving the success of the state-of-the-art CP solver *OR-Tools* [45].

Since the RCPSP is a temporal scheduling problem, any declarative representation, including ASP encodings, must explicitly or implicitly represent time. In plain ASP, the time is represented explicitly as a finite subset of natural numbers. Due to expressivity gains of DL and finite domain constraints, the time values can be left implicit in these ASP extensions. The chosen representation of time influences how the problem constraints are formulated. Although an implicit time representation is generally preferable from both modeling and computational perspectives, in this work we observe the opposite when modeling the RCPSP in ASP. Specifically, resource constraints are more convenient to express using an explicit time representation. This is because the resource capacity limits apply at each point in time, and extracting specific events of interest to substitute explicit time points incurs some extra modeling effort for ASP modulo DL and CASP formulations. Such inherent differences in the modeling approaches impact the computational performance, where our empirical comparison on small- to medium-scale RCPSP instances yields that plain ASP's simplicity outweighs its larger problem representation size, which comes as a somewhat surprising difference to usual performance trends on temporal scheduling problems [46, 47].

The main contributions of this work are twofold:

- We propose encodings for the RCPSP in ASP, ASP modulo DL, and CASP, and analyze them from a modeling perspective.
- We empirically compare our RCPSP modeling approaches on small- to medium-scale RCPSP instances [48], where we incorporate the state-of-the-art CP solver *OR-Tools* as a baseline.

The remainder of the paper is structured as follows. We start by providing preliminaries for this work in Section 2. Section 3 elaborates our plain ASP, ASP modulo DL, and CASP formulations of the RCPSP. In Section 4, we present our experimental setup and results. Finally, Section 5 concludes the paper by discussing the main findings and outlining future work.

2. Preliminaries

We first provide a brief overview of ASP, ASP modulo DL, and CASP from a problem modeling perspective, following the presentation of [47]. Second, we introduce and illustrate the RCPSP.

2.1. Answer Set Programming

ASP [23] is a knowledge representation and reasoning paradigm geared towards the modeling and solving of combinatorial (optimization) problems. A (first-order) ASP program consists of *rules* of the form $\mathtt{h}$:- $\mathtt{b}_1, \ldots, \mathtt{b}_n$., in which the head $\mathtt{h}$ is an atom $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_m)$ or a choice $\{\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_m)\}$, and each body literal $\mathtt{b}_i$ is an atom $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_m)$, possibly preceded by the default negation connective *not* and/or followed by a condition : $\mathtt{c}_1, \ldots, \mathtt{c}_l$, a built-in comparison $\mathtt{t}_1 \circ \mathtt{t}_2$ with $\circ \in \{<, <=, =, !=, >=, >\}$, or an aggregate $\mathtt{\#sum}\{\mathtt{t}_1, \ldots, \mathtt{t}_m : \mathtt{c}_1, \ldots, \mathtt{c}_l\} \circ \mathtt{t}_{m+1}$. Each $\mathtt{t}_j$ denotes a term, i.e., a constant, variable, tuple, or arithmetic expression, and each element $\mathtt{c}_k$ of a condition is an atom that may be preceded by *not* or a built-in comparison. The head $\mathtt{h}$ or body $\mathtt{b}_1, \ldots, \mathtt{b}_n$ may be omitted, in which case we call the rule an *integrity constraint* or *fact*, respectively. Roughly speaking, an ASP program is a shorthand for its ground instantiation, obtainable by substituting variables with all of the available constants and evaluating arithmetic expressions, and the semantics is given by *answer sets*, i.e., sets of (true) ground atoms such that all rules of the ground instantiation are satisfied and allow for deriving each of the ground atoms by some rule whose body is satisfied. The ground instantiation process is detailed in [49], and the answer set semantics is further elaborated in [50, 23].

ASP modulo DL integrates DL constraints [51], i.e., expressions written as $\&\textsc{diff}\{\mathtt{t}_1 - \mathtt{t}_2\} <= \mathtt{t}_3$, in the heads and bodies of rules. Except for integers, which denote their numerical values, the terms $\mathtt{t}_1$ and $\mathtt{t}_2$ represent DL variables that can be assigned any integer value. However, the difference $\mathtt{t}_1 - \mathtt{t}_2$ must not exceed the integer $\mathtt{t}_3$ for the DL constraint to be satisfied. That is, the evaluation of DL constraints interacts with the encompassing rules, and the *clingo*[DL] system [25] extends *clingo* by assuring the consistency of DL constraints imposed by an answer set. If these DL constraints are satisfiable, a canonical assignment of smallest feasible integer values to DL variables can be determined in polynomial time and is output along with the answer set by *clingo*[DL].

Finally, *clingcon* [31] extends the ASP syntax with expressions of the form $\&\textsc{sum}\{l_1 * x_1 ; l_2 * x_2 ; \ldots ; l_n * x_n\} <= k$ to represent linear constraints $l_1x_1 + l_2x_2 + \ldots + l_nx_n \leq k$, where each l_i is an integer, each x_i is a variable, and k is an arithmetic expression over integers and variables. Moreover, atoms of the form $\&\textsc{dom}\{l_1 .. l_2\} = x$ allow for limiting the domain of a variable x to the range $[l_1, l_2]$. Different from *clingo*[DL], it is also possible to include $\&\textsc{minimize}$ statements referring to the sum of the contained variables' values. The main differences between *clingo*[DL] and *clingcon* are outlined in Table 1. While *clingcon*'s linear constraints permit modeling complex problems, the consistency of DL constraints can be decided in polynomial time by *clingo*[DL]. Moreover, all theory atoms of *clingcon* are *external* and do thus not need any justification to be relevant to an answer set, while *clingo*[DL] associates this meaning to theory atoms in rule bodies only.

Table 1
Overview of *clingo*[DL] and *clingcon* concepts

Concept	*clingo*[DL]	*clingcon*
Constraint type	$x - y \leq k$	$l_1x_1 + \cdots + l_nx_n \leq k$
Theory complexity	Polynomial-time decidable	NP-complete
Theory atoms	Head atoms defined; body atoms external	All theory atoms are external
Syntax	`&DIFF{x - y} <= 5`	`&SUM{3*x; 4*y} <= z-7`

2.2. Resource-Constrained Project Scheduling Problem (RCPSP)

The RCPSP is a canonical combinatorial optimization problem that formally captures the challenges of scheduling under both precedence and resource constraints [12, 9]. It is known to be NP-hard in the strong sense [12, 52] and has been extensively studied in the operations research and project management literature.

Formally, the RCPSP is defined on a project network represented as a directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \ldots, n\}$ is a set of activities and $\mathcal{E}$ is a set of precedence relations. The problem is further characterized by the following concepts:

Activities. Each activity $i \in \mathcal{V}$ has an associated duration $d_i \geq 0$ and must be executed without interruption once started — a condition known as the *non-preemption* assumption.

Precedence Constraints. For each arc $(i, j) \in \mathcal{E}$, activity j cannot begin before activity i has been completed. Denoting the start times of the activities by S_i and S_j, this is expressed as:

$$S_j \geq S_i + d_i, \quad \forall (i, j) \in \mathcal{E}$$

Renewable Resources. The project involves a set $\mathcal{R} = \{1, \ldots, m\}$ of renewable resources, where each resource $k \in \mathcal{R}$ has a fixed capacity R_k available at every time period. Each activity $i \in \mathcal{V}$ requires $r_{ik} \geq 0$ units of resource k over its entire duration. At any time point t, the total resource consumption by all activities in execution must not exceed the available capacity:

$$\sum_{i \in \mathcal{A}(t)} r_{ik} \leq R_k, \quad \forall k \in \mathcal{R}, \ \forall t$$

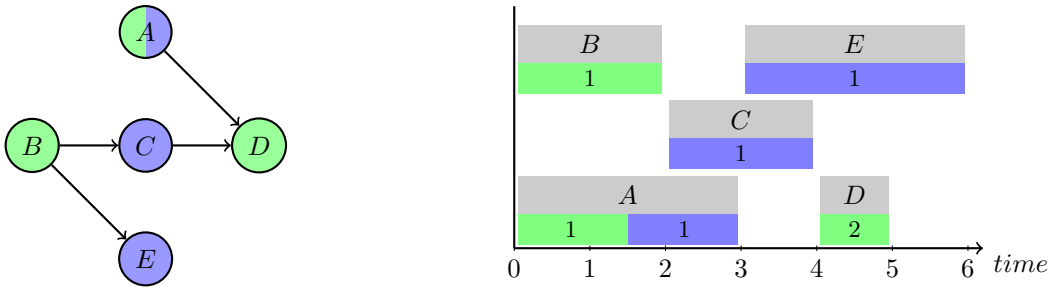
where $\mathcal{A}(t) = \{i \in \mathcal{V} \mid S_i \leq t < S_i + d_i\}$ denotes the set of activities being executed at time t.

Objective. The standard objective of the RCPSP is to find a feasible schedule — that is, an assignment of start times $\{S_i\}_{i \in \mathcal{V}}$ to all activities — such that both the precedence constraints and the resource constraints are satisfied, while minimizing the project makespan $C_{\max} = \max\{S_i + d_i \mid i \in \mathcal{V}\}$, i.e., the latest completion time of any activity.

Example 1. Consider a project consisting of five activities, labeled A, B, C, D, E, and two resources, *blue* and *green*, both having a capacity of 2 units. Activities A and C precede D, and B precedes C and E. Activities B and D require the green resource in quantities 1 or 2, respectively, activities C and E require the blue resource in quantity 1, and activity A requires 1 unit of each resource. Activity D has a duration of 1 time unit, activities B and C have a duration of 2, and A and E have a duration of 3.

The project network is often visually depicted as a dependency graph, as presented in Figure 1(a). An optimal schedule for this project is shown in Figure 1(b). Here, the x-axis displays time. Activities are represented by gray rectangles including their labels, and the colored rectangles below them indicate resource requirements in terms of the resources' colors and units. Vertical activity stacking is purely for visualization, where activities that overlap at time points are executed simultaneously.

This example is designed to point out the following properties that will turn out to be important from the aspect of modeling:



(a) Project network (precedence relations) (b) Optimal schedule

Figure 1: An example RCPSP instance

```
task(a,3).   res(a,r1,1). res(a,r2,1).   prec(a,d).              capacity(r1,2).
task(b,2).   res(b,r1,1).                prec(b,c). prec(b,e).   capacity(r2,2).
task(c,2).                res(c,r2,1).   prec(c,d).
task(d,1).   res(d,r1,2).
task(e,3).                res(e,r2,1).
```

Listing 1: Facts representation of the RCPSP instance in Example 1

- Merely accumulating the resource requirements of activities that overlap at some time is insufficient to determine resource consumption. For example, activities A and C overlap, and likewise C and E, for a total of 3 activities that require the blue resource. However, only two of the activities are executed simultaneously. This means that the maximum use of the blue resource is 2, not 3.
- Regarding the availability of resources, activity D could start earlier, i.e., at time point 3 rather than 4. However, the precedence constraints necessitate that D is executed only after the completion of C. Conversely, activity E could start earlier from the perspective of its precedence relation with B, but due the blue resource limit, it needs to wait for 1 resource unit to get free.
- Figure 1(b) displays one of the optimal schedules. An alternative optimal schedule can be obtained by delaying the execution of activities C and D by 1 time unit each. In total, our example RCPSP instance has 5 optimal schedules with a project makespan of 6.

3. Modeling

As customary in ASP, our RCPSP models combine instance-specific facts with a general encoding part. We start by describing the fact format, then present our plain ASP encoding, and finally turn to ASP modulo DL and CASP encodings circumventing the explicit representation of time points.

3.1. Fact Format

The format of facts specifying an RCPSP instance, such as the one described in Example 1, is illustrated by Listing 1. Activities together with their durations are given by the `task`/2 predicate. The `res`/3 predicate specifies the activities' resource requirements, where `r1` stands for the green and `r2` for the blue resource according to the coloring scheme of Figure 1(a). Precedence relations between the activities are expressed by the `prec`/2 predicate, and the maximum capacity of 2 units for both resources, `r1` and `r2`, is given by the `capacity`/2 predicate.

3.2. Plain ASP Encoding

To begin with, the rules in Listing 2 define domain predicates that are deterministically evaluated in the grounding phase. This includes `task`/1 to access activities' labels whenever their durations do not matter (line 1), `first`/1 for activities that have no predecessors in the precedence relations (line 2), and

```
1  % all activities and activities without any predecessors or successors, respectively
2  task(I)  :- task(I,D).
3  first(I) :- task(I), not prec(_,I).
4  final(I) :- task(I), not prec(I,_).
5
6  % reflexive transitive closure of precedence among activities
7  trans(I1,I1) :- task(I1).
8  trans(I1,I3) :- task(I1), prec(I1,I2), trans(I2,I3).
9
10 % activities I1 and I2 without precedence may use a common resource R simultaneously
11 check(I1,I2,R) :- res(I1,R,X1), res(I2,R,X2), capacity(R,C),
12                  not trans(I1,I2), not trans(I2,I1).
13 check(I1,I2)   :- check(I1,I2,R).
14
15 % sum of activity durations provides upper bound on makespan and variable domains
16 horizon(H)    :- #sum{D,I : task(I,D)} = H.
17 range(0..H-1) :- horizon(H).
```

Listing 2: Domain predicates used by RCPSP encodings

```
19 % undone(I,T) expresses that activity I is not yet completed at time T
20  undone(I,T)  :- task(I,D), range(T), T < D.
21 {undone(I,T)} :- undone(I,T-1), range(T).
22
23 % successor activity J takes at least its duration D from completing predecessor I
24 :- undone(I,T), prec(I,J), task(J,D), T' = T+D, range(T'-1), not undone(J,T').
25
26 % resource capacities must not be exceeded at any time T
27 :- capacity(R,C), range(T),
28   #sum{X,I : res(I,R,X), task(I,D), undone(I,T), not undone(I,T+D)} > C.
29
30 % the times T at which activities without successors aren't completed yield makespan
31 :~ final(I), undone(I,T). [1,T]
```

Listing 3: Plain ASP encoding of RCPSP

similarly `final`/1 for activities without successors (line 3). The `trans`/2 predicate, defined by the rules in lines 7 and 8, provides the reflexive transitive closure of the precedence relations. Note that neither `trans`(i,j) nor `trans`(j,i) holds for (distinct) activities $i,j \in \mathcal{V}$ that may be executed simultaneously. If such activities have a required resource in common, it is indicated by the `check`/3 predicate (lines 11-12), and `check`/2 is derived by projection omitting the shared resource (line 13). Moreover, the sum of activities' durations, calculated by the rule in line 16, constitutes a (trivial) upper bound on the project makespan, which is provided by the `horizon`/1 predicate. Accordingly, the `range`/1 predicate gathers all smaller time points at which some activity may still be executed.

The second part of our plain ASP encoding, shown in Listing 3, takes care of generating feasible schedules of minimal project makespan. For each activity $i \in \mathcal{V}$, an answer set includes atoms of the form `undone`$(i,0)$, `undone`$(i,1)$, ..., `undone`$(i,t-2)$, `undone`$(i,t-1)$ to represent that i gets completed at the time point t. In turn, activity i is executed at the time points ranging from $t-d_i$ to $t-1$, where d_i is the duration of i. Given that i cannot be completed before the time point d_i, `undone`/2 atoms up to d_i-1 are unconditionally derived by the rule in line 20, while the choice rule in line 21 allows for delaying the completion time, as long as the upper bound given by `horizon`/1 is not exceeded.

Based on atoms `undone`(i,t) for a predecessor i of the activity j, the integrity constraint in line 24 requires `undone`$(j,t+d_j)$ to hold, i.e., j takes at least its duration d_j to complete after i. Ground instances of this integrity constraint need only be produced up to $t+d_j$ matching the upper bound

of horizon/1, since $\texttt{undone}(j,t+d_j)$ cannot hold for this threshold. The integrity constraint in lines 27-28 models resource capacity checks at each time point t, where the resource requirements of all activities i that are neither completed at t nor yet uncompleted at $t+d_i$, i.e., not yet started at t, are added up and compared against the resource capacity. This guarantees that resource capacities are never exceeded and schedules with the start times $S_i = t+1-d_i$, for the greatest time point t such that $\texttt{undone}(i,t)$ belongs to an answer set, are feasible. Finally, makespan minimization is expressed by the weak constraint in line 31, which penalizes each time point at which some activity without successors is yet uncompleted by the uniform weight 1. Activities that have successors can be disregarded here because their completion times will certainly be subsumed by a successor activity executed later on.

The following result establishes the soundness and completeness of our plain ASP encoding of RCPSP.

Theorem 1. Given any RCPSP $G = (\mathcal{V}, \mathcal{E})$ along with a schedule $\{S_i\}_{i\in\mathcal{V}}$, we have that $\{S_i\}_{i\in\mathcal{V}}$ is a feasible schedules of minimal project makespan $C_{\max}$ iff the rules in Listings 2 and 3 yield an optimal answer set X such that $\{(i,t) \mid \texttt{undone}(i,t) \in X\} = \{(i,t) \mid i \in \mathcal{V}, 0 \leq t < S_i + d_i\}$ on the fact representation of G.

Proof. The rules in Listing 2 derive a unique answer set X' on the fact representation of G, and $X' \subseteq X \subseteq X' \cup \{\texttt{undone}(i,t) \mid i \in \mathcal{V}, 0 \leq t < \sum_{i\in\mathcal{V}} d_i\}$ holds for any answer set X obtained for the rules in Listings 2 and 3 on the fact representation of G. In the following, we consider the set $X = X' \cup \{\texttt{undone}(i,t) \mid i \in \mathcal{V}, 0 \leq t < S_i + d_i\}$ of atoms.

$\Leftarrow$: Assume that X is an answer set for the rules in Listings 2 and 3 on the fact representation of G. Then, the integrity constraints in lines 24 and 27-28 check that $\{S_i\}_{i\in\mathcal{V}}$ satisfies the precedence and resource constraints, so that $\{S_i\}_{i\in\mathcal{V}}$ is a feasible schedule. Moreover, if X is optimal, for any schedule $\{S'_i\}_{i\in\mathcal{V}}$ such that $C_{\max} > \max\{S'_i + d_i \mid i \in \mathcal{V}\}$, the fact that $X' \cup \{\texttt{undone}(i,t) \mid i \in \mathcal{V}, 0 \leq t < S'_i + d_i\}$ is not a (better) answer set, which would strictly reduce the penalized ground instances of the weak constraint in line 31, means that $\{S'_i\}_{i\in\mathcal{V}}$ does not satisfy the precedence and resource constraints. Hence, the optimality of X yields that the project makespan $C_{\max}$ is minimal.

$\Rightarrow$: Assume that X is not an optimal answer set for the rules in Listings 2 and 3 on the fact representation of G. Then, some of the following cases applies:

1. If $\sum_{i\in\mathcal{V}} d_i < C_{\max}$, there is at least one time point smaller than $C_{\max}$ at which no activity is executed. That is, $C_{\max}$ cannot be the minimal project makespan.
2. If $\sum_{i\in\mathcal{V}} d_i \geq C_{\max}$ and X does not satisfy some ground instance of the integrity constraint in line 24, there is an arc $(i,j) \in \mathcal{E}$ such that $S_j < S_i + d_i$. That is, $\{S_i\}_{i\in\mathcal{V}}$ does not satisfy the precedence constraints.
3. If $\sum_{i\in\mathcal{V}} d_i \geq C_{\max}$ and X does not satisfy some ground instance of the integrity constraint in lines 27-28, there is a time point t such that $\sum_{i\in\mathcal{A}(t)} r_{ik} > R_k$ holds for some resource $k \in \mathcal{R}$. That is, $\{S_i\}_{i\in\mathcal{V}}$ does not satisfy the resource constraints.
4. If neither of the previous cases applies, we have that X is not optimal and there is an (optimal) answer set Y for which $C_{\max} > \max\{t+1 \mid \texttt{undone}(i,t) \in Y\}$. By the "$\Leftarrow$" direction, $\{S'_i = t_i - d_i\}_{i\in\mathcal{V}, t_i = \max\{t+1 \mid \texttt{undone}(i,t)\in Y\}}$ is a feasible schedule such that $C_{\max} > \max\{S'_i + d_i \mid i \in \mathcal{V}\}$.

□

A limiting factor regarding the scalability of our plain ASP encoding is its explicit representation of time points, which get materialized in the grounding phase. However, the availability of `#sum` aggregates allows us to represent resource capacity checks at each time point compactly, thanks to a pseudo-Boolean constraint formulation rather than the clause format of (maximum) satisfiability [53].

3.3. ASP Modulo DL and CASP Encodings

The primary goal of incorporating integer variables as first-class citizens in ASP modulo DL and CASP is to reduce the ground instantiation size. For RCPSP, we make use of integer variables of the form (i,d_i) to refer to the completion time of an activity i of duration d_i without materializing all candidate

```
19 % duration D is lower bound for completing activity I without predecessors
20 &DIFF{    0 - (I,D) } <= -D :- task(I,D), first(I).
21 % completing a successor activity J after I takes at least its duration D'
22 &DIFF{(I,D) - (J,D')} <= -D' :- prec(I,J), task(I,D), task(J,D').
23
24 % case 1 for ignore(I,J): J gets completed before I starts
25 ignore(I,J) :- check(I,J), task(I,D), task(J,D'), &DIFF{(J,D') - (I,D)} <= -D.
26 % case 2 for ignore(I,J): I gets completed before J
27 ignore(I,J) :- check(I,J), task(I,D), task(J,D'), &DIFF{(I,D) - (J,D')} <= -1.
28
29 % check resource capacity for activity J and all I executed at its completion time
30 :- res(J,R,X'), capacity(R,C),
31   #sum{X,I : check(I,J,R), res(I,R,X), not ignore(I,J)} > C-X'.
32
33 % completion times of activities without successors provide lower bounds on makespan
34 &DIFF{(I,D) - makespan} <= 0 :- task(I,D), final(I).
```

Listing 4: ASP modulo DL encoding of RCPSP using `--minimize-variable=makespan`

values in the grounding phase. Such an integer variable replaces the entire collection of $\texttt{undone}(i,t)$ atoms for all time points t in range.

In more detail, the DL constraint in line 20 of Listing 4 states that the duration d_i is the smallest possible completion time of an activity i without predecessors. These lower bounds are carried on to successor activities j by adding the respective duration d_j to the integer variable (i,d_i) in line 22, given that the expression $(i,d_i) - (j,d_j) \leq -d_j$ is equivalent to $(i,d_i) + d_j \leq (j,d_j)$.

Lines 24-31 constitute the most sophisticated part of our ASP modulo DL encoding, where resource capacity checks are expressed without explicitly referencing time points. In the classical Job-shop Scheduling Problem (JSP) [54, 55, 47], the rule in line 25 (in combination with an integrity constraint `:-check(I,J),not ignore(I,J),not ignore(J,I).`) would be sufficient to make sure that the execution of (distinct) activities $i, j \in \mathcal{V}$ sharing a common resource is non-overlapping. In the setting of RCPSP, however, the activities can be executed simultaneously when the resource capacities are not exceeded. Hence, the non-overlapping case in line 25, based on the condition $(j,d_j) - (i,d_i) \leq -d_i$ or $(j,d_j) + d_i \leq (i,d_i)$, respectively, may not apply to i and j (and also not to j and i in inverse order). For example, the optimal schedule in Figure 1(b) is such that the execution of activity C overlaps both A and E, each of which utilize (1 unit of) the blue resource. Since two units are the capacity limit, the optimal schedule would be mistakenly discarded if non-overlapping were the sole criterion for capacity checks. In fact, the schedule is feasible because activity A gets completed before E is started, and the general observation is that, at each activity's completion time, at most one other activity requiring the blue resource is executed simultaneously. We adopt this condition and encode capacity checks at activities' completion, serving as events that substitute explicit time points of the plain ASP encoding.

Accordingly, the case that activity j gets completed before i starts is augmented in line 27 by also deriving $\texttt{ignore}(i,j)$ when i gets completed before j, as expressed by the condition $(i,d_i) - (j,d_j) \leq -1$ or $(i,d_i) + 1 \leq (j,d_j)$, respectively. Putting these two conditions together, $\texttt{ignore}(i,j)$ holds iff an activity i that shares some resource with j is not currently executed at the completion time of j. The integrity constraint in lines 30-31 takes the remaining capacity, i.e., the capacity limit reduced by j's requirement, as threshold for other activities i executed simultaneously to the completion of j. This provides an alternative formulation, compared to the plain ASP encoding, of resource capacity checks.

The final constituent is makespan minimization by means of an integer variable `makespan`, whose value is in line 34 bound from below by the completion times of activities i without successors, considering that the expression $(i,d_i) - \texttt{makespan} \leq 0$ is equivalent to $(i,d_i) \leq \texttt{makespan}$. We again note that the completion times of activities that have successors are subsumed, and the option `--minimize-variable=makespan` instructs *clingo*[DL] to minimize the `makespan` variable's value.

The CASP encoding in Listing 5 is formulated in the input language of *clingcon*, where the rules in

```
19 % duration D is lower bound for completing activity I without predecessors
20 &sum{D} <= (I,D)          :- task(I,D), first(I).
21 % completing a successor activity J after I takes at least its duration D'
22 &sum{(I,D); D'} <= (J,D') :- prec(I,J), task(I,D), task(J,D').
23
24 % case 1 for ignore(I,J): J gets completed before I starts
25 ignore(I,J) :- check(I,J), task(I,D), task(J,D'), &sum{(J,D'); D} <= (I,D).
26 % case 2 for ignore(I,J): I gets completed before J
27 ignore(I,J) :- check(I,J), task(I,D), task(J,D'), &sum{(I,D)} < (J,D').
28
29 % check resource capacity for activity J and all I executed at its completion time
30 :- res(J,R,X'), capacity(R,C),
31   #sum{X,I : check(I,J,R), res(I,R,X), not ignore(I,J)} > C-X'.
32
33 % completion times of activities without successors provide lower bounds on makespan
34 &sum{(I,D)} <= makespan :- task(I,D), final(I).
35
36 % miminize the makespan (variable's value)
37 &minimize{makespan}.
38
39 % activities' completion times and makespan are restricted by horizon
40 &dom{D..H} = (I,D)    :- task(I,D), horizon(H).
41 &dom{0..H} = makespan :- horizon(H).
```

Listing 5: CASP encoding of RCPSP

lines 19-34 are mere syntactic mappings from our ASP modulo DL encoding, using *clingcon*'s &sum constraints in place of &diff atoms. Likewise, the &minimize statement in line 37 takes the role of *clingo*[DL]'s `--minimize-variable=makespan` option. Given that the `horizon/1` predicate provides a (trivial) upper bound on the project makespan, the rules in lines 40 and 41 additionally define &dom atoms limiting integer variables for activities' completion times and `makespan` to their feasible domains.

Soundness and completeness according to Theorem 1 also hold for our ASP modulo DL and CASP encodings.

Proposition 1. Theorem 1 remains valid for either Listing 4 or Listing 5 in place of Listing 3 and integer variable assignments $(i, d_i) = S_i + d_i$ instead of the condition $\{(i,t) \mid \texttt{undone}(i,t) \in X\} = \{(i,t) \mid i \in \mathcal{V}, 0 \leq t < S_i + d_i\}$ on atoms belonging to an optimal answer set X.

Proof. We show that Theorem 1 remains valid when the integrity constraint in lines 27-28 of Listing 3 is replaced by the following rules:

```
% case 1 for ignore(I,J): J gets completed before I starts
ignore(I,J) :- check(I,J), task(I,D), undone(I,T-1), not undone(J,T-D), D < T.
% case 2 for ignore(I,J): I gets completed before J
ignore(I,J) :- check(I,J), undone(J,T), not undone(I,T).

% check resource capacity for activity J and all I executed at its completion time
:- res(J,R,X'), capacity(R,C),
  #sum{X,I : check(I,J,R), res(I,R,X), not ignore(I,J)} > C-X'.
```

For (distinct) activities $i, j \in \mathcal{V}$ that are not connected by any directed path in G and share some resource $k \in \mathcal{R}$, i.e., $r_{ik} > 0$ and $r_{jk} > 0$ hold, $\texttt{ignore}(i,j)$ is derived iff $S_i \geq S_j + d_j$ or $S_j + d_j > S_i + d_i$ applies. In turn, $\texttt{ignore}(i,j)$ remains underivable iff we have that $S_i < S_j + d_j \leq S_i + d_i$, i.e., the execution of activity i has started and is not completed before the completion of j. Hence, the integrity constraint, which matches the one in lines 30-31 of Listing 4 and Listing 5, makes sure that the accumulated requirements r_{ik} of such simultaneously executed activities i do not exceed $R_k - r_{jk}$ at the completion

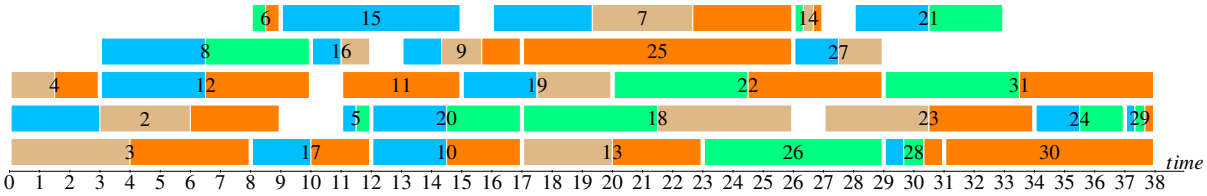


Figure 2: Visualization of the benchmark instance J3024_8

time of j for shared resources $k \in \mathcal{R}$. As the integrity constraint in lines 27-28 of Listing 3 guarantees that resource capacities are never exceeded, any answer set X obtained for the rules in Listings 2 and 3 on the fact representation of G yields $X \cup \{\texttt{ignore}(i,j) \mid \texttt{check}(i,j) \in X, S_i \geq S_j + d_j \text{ or } S_j + d_j > S_i + d_i\}$ as an answer set when the integrity constraint is replaced by the above rules. Conversely, an answer set X obtained with the above rules satisfies the integrity constraint in lines 27-28 of Listing 3, as establishing $\sum_{i \in \mathcal{A}(t)} r_{ik} \leq R_k$ at time points $t = S_j + d_j - 1$, for activities $j \in \mathcal{V}$ with $r_{jk} > 0$, suffices.

Given these considerations, we conclude that the ASP modulo DL and CASP encodings in Listing 4 and Listing 5 are sound and complete, as they syntactically reformulate Listing 3, with the above rules replacing the integrity constraint in lines 27-28, in terms of ASP modulo DL or CASP, respectively. □

4. Experiments

We evaluate our RCPSP modeling approaches on well-known benchmark sets [48] comprising 2,040 small- to medium-scale instances, namely, three times 480 instances with 30, 60, and 90 activities and another 600 instances with 600 activities.[1] For illustration, Figure 2 visualizes an optimal schedule for one of the RCPSP instances with 30 activities and 4 resources, where colors indicate the required resources. As a specific observation, activity 11 starts at the time point 11 although no other activity (which could be a predecessor or share a common resource) is completed at time 11. Hence, the activity could safely start at time point 10 already, yet without reducing the optimal project makespan. In preliminary experiments, we also investigated hard constraints to eliminate such unnecessary delays, yet observed a deterioration of the solving performance due to increasing the ground instantiation size by about one order of magnitude. Moreover, we note that the makespan 38 matches the (trivial) lower bound given by a longest sequence of activities according to the precedence constraints, while this easy optimality condition does not apply to the majority of RCPSP instances, for which optimality proofs remain hard.

In our experiments, we ran *clingo* v5.8.0 on the plain ASP encoding as well as a plain ASP formulation mimicking the ASP modulo DL encoding idea of resource capacity checks at activities' completion times (denoted by ASP_DL-like in Table 2), *clingo*[DL] v1.5.0 on the ASP modulo DL encoding (abbreviated by DL in Table 2), and *clingcon* v5.2.1 on the CASP encoding. For comparison, we also ran the state-of-the-art CP solver *OR-Tools* v9.15 on an RCPSP model featuring interval variables and global `cumulative` constraints.[2] Except for *OR-Tools*, where we determined the option `linearization_level:0` as most effective in preliminary experiments, the solvers were run in their default configurations, which means that all of them perform model-guided optimization [56]. While core-guided optimization strategies are available as well, they turned out as significantly less effective and are thus only applied within parallel portfolios (tested in Table 4). All experiments were run on an *Intel® Core™ i7-8650U CPU Dell Latitude 5590* machine under *Windows 10.1*, imposing a timeout of 900 seconds for each instance.

Table 2 compares the solving performance with the five encodings on the small-scale benchmark set with 30 activities. The second column reports the number of instances for which a proven optimal schedule has been obtained within the given 900 seconds, the third column expresses the optimality

[1] The experimental setup and all details needed for reproducing the presented results are available at https://github.com/prosysscience/RCPSP-ASPOCP2026.

[2] The source code is made available in the official *OR-Tools* repository, and can be found at https://raw.githubusercontent.com/google/or-tools/refs/heads/stable/examples/python/rcpsp_sat.py.

Table 2
Comparison of different encodings on the benchmark set with 30 activities

Encoding	Optimal	Rate	Time (s)
ASP	**480**	**100.0**	**90.6**
ASP_DL-like	472	98.3	9,676.9
DL	475	98.9	6,535.8
CASP	**480**	**100.0**	219.5
OR-Tools	**480**	**100.0**	527.9

Table 3
Comparison of different encodings on the benchmark sets with 60, 90, and 120 activities

Activities	Encoding	Optimal	Rate	Time (s)
60	ASP	**439**	**91.5**	**40,373.7**
	CASP	428	89.2	55,383.4
	OR-Tools	435	90.6	46,426.2
90	ASP	**406**	**84.6**	**70,470.5**
	CASP	393	81.9	94,649.1
	OR-Tools	399	83.1	78,219.4
120	ASP	**289**	**48.2**	**282,284.6**
	CASP	271	45.2	317,096.7
	OR-Tools	287	47.8	296,738.5

rate as percentage of the total number of 480 instances solved to optimality, and the fourth column reports the cumulative solving time in seconds over all instances. We observe that three encodings achieve a perfect optimality rate of 100%, namely, ASP, CASP, and *OR-Tools*. Regarding computational efficiency, ASP is most performant by requiring only 90.6 seconds in total, followed by CASP with 219.5 seconds, and then *OR-Tools* with 527.9 seconds. The remaining encodings, ASP_DL-like and DL, yield proven optimal schedules for 472 (98.3%) or 475 (98.9%) instances, respectively. Although this shows that *clingo*[DL] is outperformed by *clingcon* with an equivalent encoding, *clingo*[DL] is still significantly ahead of *clingo* with the ASP_DL-like encoding, which emphasizes that the reduced problem representation size thanks to integer variables is beneficial when using the same encoding idea. However, for the medium-scale instances with 60, 90, and 120 activities, we restrict ourselves to comparing ASP, CASP, and *OR-Tools*, while dropping the dominated ASP_DL-like and DL encodings.

Across all three of the benchmark sets with 60, 90, and 120 activities compared in Table 3, ASP consistently achieves the highest optimality rate and the shortest cumulative solving time, establishing it as the best-performing encoding overall. We also observe that *OR-Tools* performs second best and reverses the previous trend of a longer cumulative solving time than CASP on the small-scale instances. This may indicate a greater constant overhead of running *OR-Tools*, which becomes less pronounced as the instance size and hardness grow. In fact, as the instance size grows, the optimality rates decline with all encodings, reflecting the increased hardness of the larger instances. In preliminary experiments, we additionally tested whether *clingo*'s `--trans-ext` option, turning `#sum` aggregates into clause format by means of the sequential weight counter encoding [57, 58], can make the optimization more effective in exchange for a larger problem representation size. The effect was clearly negative, as the size overhead does not bring about significant savings of search efforts, which would be required to reduce the overall solving time. We thus conclude that the pseudo-Boolean constraints expressed by `#sum` aggregates provide adequate means to model resource capacity checks based on explicitly represented time points.

Table 4 compares the performance of *clingo* with the plain ASP encoding and *OR-Tools* in two solving modes: running the solvers' (default) portfolios with 8 differently configured search threads in parallel, and the previously used single-threading mode for which the numbers of proven optimal schedules are reproduced in parentheses. The results show that increasing the number of threads and processors

Table 4
Comparison of multi-threaded ASP solving and *OR-Tools* on the benchmark sets with 30, 60, 90, and 120 activities

Encoding	30	60	90	120
ASP	**480** (**480**)	438 (**439**)	**407** (406)	**293** (289)
OR-Tools	**480** (**480**)	433 (435)	398 (399)	285 (287)

from 1 to 8 yields no significant improvements in the number of instances solved to optimality for either *clingo* or *OR-Tools*, indicating that parallelism has a negligible impact on optimality proofs within the given 900 seconds wall clock time. That is, *clingo* has an edge on *OR-Tools* in both the single- and the multi-threaded solving mode, confirming that the advantage of our plain ASP encoding over finite domain constraint formulations is consistent and independent of the allocated computational resources.

5. Conclusion

We have presented and evaluated several modeling approaches for encoding the RCPSP. In contrast to previous investigations on the classical JSP [47], where *OR-Tools*' interval variables and global constraints [59] give it an advantage on *clingo*[DL], while the problem representation size is entirely prohibitive for plain ASP approaches, we find that *clingo* with our plain ASP encoding benefits from the explicit representation of time points and most effectively proves optimal schedules for small- to medium-scale RCPSP instances. Among the compared approaches incorporating DL or finite domain constraints, *OR-Tools* comes closest to *clingo*, and *clingcon* performs better than *clingo*[DL] (and *clingo*) with a more sophisticated encoding of capacity checks at specific events instead of all time points in range. Certainly, the time range impacts the scalability of our plain ASP encoding, and *clingo* gets along well with the comparably moderate size of RCPSP instances in the used benchmark set [48]. However, `#sum` aggregates or pseudo-Boolean constraints, respectively, turn out to be advantageous for representing resource capacity limits compactly, and going to the clause format of (maximum) satisfiability would likely introduce more overhead than can be compensated by means of more performant solving methods.

Despite their moderate size, a significant portion of instances in the benchmark set [48] are highly combinatorial, given that quite many of those with 60, 90, or 120 activities could not be solved to optimality yet.[3] While finding some feasible schedule is easy to accomplish for each instance, gradually improving the makespan and completing the final optimality proof can be extraordinarily hard. We observed that the solving performance heavily varies between seemingly similar instances of the same size, taking just a few seconds or virtual infinity with the same problem encoding and solver. Future work includes a more detailed investigation of instance characteristics and specific properties of optimal schedules, which may be available in the literature, in order to identify so-called redundant constraints with the potential to boost optimality proofs. Straightforward constraints, like prohibiting unnecessary delayed starts of activities, merely increase the problem representation size without bringing about significant search savings, so that more elaborate concepts are needed.

Acknowledgments

This research was funded by the Austrian Science Fund (FWF) project 10.55776/PAT1599524. We are grateful to Ulrich Junker, Roland Kaminski, Benjamin Kaufmann, Valentin Mayer-Eichberger, Ryan Murphy, Max Ostrowski, Javier Romero, Torsten Schaub, Philipp Wanko, and the anonymous reviewers for their valuable comments and suggestions, helping to improve this work.

[3]Best known solutions and the times taken to compute them are provided at: http://solutionsupdate.ugent.be/index.php/rcpsp

Declaration on Generative AI

During the writing of this paper, the authors used Microsoft Copilot as an editor add-on making text suggestions, and Grammarly was used to improve the text quality. All contents and formulations are the authors' original work. No parts of the paper were generated by prompting, and any AI-made suggestions have been carefully checked and thoroughly revised.

References

[1] S. H. Dike, Project scheduling with resource constraints, IEEE Transactions on Engineering Management EM-11 (1964) 155–157. doi:10.1109/TEM.1964.6446423.

[2] G. Simonin, C. Artigues, E. Hebrard, P. Lopez, Scheduling scientific experiments on the Rosetta/Philae mission, in: M. Milano (Ed.), Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'12), volume 7514 of *Lecture Notes in Computer Science*, Springer-Verlag, 2012, pp. 23–37. doi:10.1007/978-3-642-33558-7_5.

[3] T. Borreguero Sanchidrián, T. Portoleau, C. Artigues, A. García Sánchez, M. Ortega Mier, P. Lopez, Large neighborhood search for an aeronautical assembly line time-constrained scheduling problem with multiple modes and a resource leveling objective, Annals of Operations Research 338 (2024) 13–40. doi:10.1007/S10479-023-05629-3.

[4] O. Polo-Mejía, C. Artigues, P. Lopez, V. Basini, Mixed-integer/linear and constraint programming approaches for activity scheduling in a nuclear research facility, International Journal of Production Research 58 (2020) 7149–7166. doi:10.1080/00207543.2019.1693654.

[5] E. W. Davis, Resource allocation in project network models: A survey, Journal of Industrial Engineering 17 (1966) 177–188.

[6] L. Özdamar, G. Ulusoy, A survey on the resource-constrained project scheduling problem, IIE Transactions 27 (1995) 574–586. doi:10.1080/07408179508936773.

[7] J. J. Moder, C. R. Phillips, E. W. Davis, Project Management with CPM, PERT and Precedence Diagramming, Van Nostrand Reinhold, 1983.

[8] H. Kerzner, Project management: A systems approach to planning, scheduling, and controlling, John Wiley & Sons, 2025.

[9] P. Brucker, A. Drexl, R. Möhring, K. Neumann, E. Pesch, Resource-constrained project scheduling: Notation, classification, models, and methods, European Journal of Operational Research 112 (1999) 3–41. doi:10.1016/S0377-2217(98)00204-5.

[10] J. E. Kelley Jr., M. R. Walker, Critical-path planning and scheduling, in: F. E. Heart (Ed.), Proceedings of the Eastern Joint IRE-AIEE-ACM Computer Conference (IRE-AIEE-ACM'59), ACM, 1959, pp. 160–173. doi:10.1145/1460299.1460318.

[11] D. G. Malcolm, J. H. Roseboom, C. E. Clark, W. Fazar, Application of a technique for research and development program evaluation, Operations Research 7 (1959) 646–669.

[12] J. Blazewicz, J. K. Lenstra, A. H. G. R. Kan, Scheduling subject to resource constraints: Classification and complexity, Discrete Applied Mathematics 5 (1983) 11–24. doi:10.1016/0166-218X(83)90012-4.

[13] R. Kolisch, S. Hartmann, Experimental investigation of heuristics for resource-constrained project scheduling: An update, European Journal of Operational Research 174 (2006) 23–37. doi:10.1016/J.EJOR.2005.01.065.

[14] A. A. B. Pritsker, L. J. Watters, P. M. Wolfe, Multiproject scheduling with limited resources: A zero-one programming approach, Management Science 16 (1969) 93–108. doi:10.1287/mnsc.16.1.93.

[15] E. L. Demeulemeester, W. S. Herroelen, A branch-and-bound procedure for the generalized resource-constrained project scheduling problem, Operations Research 45 (1997) 201–212. doi:10.1287/opre.45.2.201.

[16] E. W. Davis, J. H. Patterson, A comparison of heuristic and optimum solutions in resource-

constrained project scheduling, Management Science 21 (1975) 944–955. doi:10.1287/mnsc.21.8.944.

[17] R. Pellerin, N. Perrier, F. Berthaut, A survey of hybrid metaheuristics for the resource-constrained project scheduling problem, European Journal of Operational Research 280 (2020) 395–416. doi:10.1016/j.ejor.2019.01.063.

[18] O. Liess, P. Michelon, A constraint programming approach for the resource-constrained project scheduling problem, Annals of Operations Research 157 (2008) 25–36. doi:10.1007/S10479-007-0188-Y.

[19] O. Koné, C. Artigues, P. Lopez, M. Mongeau, Event-based MILP models for resource-constrained project scheduling problems, Computers & Operations Research 38 (2011) 3–13. doi:10.1016/j.cor.2009.12.011.

[20] F. Hübner, P. Gerhards, C. Stürck, R. Volk, Solving the nuclear dismantling project scheduling problem by combining mixed-integer and constraint programming techniques and metaheuristics, Journal of Scheduling 24 (2021) 269–290. doi:10.1007/S10951-021-00682-X.

[21] K. M. Sallam, R. K. Chakrabortty, M. J. Ryan, A reinforcement learning based multi-method approach for stochastic resource constrained project scheduling problems, Expert Systems with Applications 169 (2021) 114479. doi:10.1016/j.eswa.2020.114479.

[22] T. Wang, W. Cheng, Y. Zhang, X. Hu, Dynamic selection of priority rules based on deep reinforcement learning for rescheduling of RCPSP, IFAC-PapersOnLine 55 (2022) 2144–2149. doi:10.1016/j.ifacol.2022.10.025.

[23] V. Lifschitz, Answer Set Programming, Springer-Verlag, 2019. doi:10.1007/978-3-030-24658-7.

[24] P. Cappanera, S. Caruso, C. Dodaro, G. Galatà, M. Gavanelli, M. Maratea, C. Marte, M. Mochi, M. Nonato, M. Roma, Recent answer set programming applications to scheduling problems in digital health, in: D. Aineto, R. De Benedictis, M. Maratea, M. Mittelmann, G. Monaco, E. Scala, L. Serafini, I. Serina, F. Spegni, E. Tosello, A. Umbrico, M. Vallati (Eds.), Proceedings of the International Workshop on Artificial Intelligence for Climate Change, Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (AI4CC-IPS-RCRA-SPIRIT'24), CEUR-WS.org, 2024.

[25] T. Janhunen, R. Kaminski, M. Ostrowski, T. Schaub, S. Schellhorn, P. Wanko, Clingo goes linear constraints over reals and integers, Theory and Practice of Logic Programming 17 (2017) 872–888. doi:10.1017/S1471068417000242.

[26] Y. Lierler, Constraint answer set programming: Integrational and translational (or SMT-based) approaches, Theory and Practice of Logic Programming 23 (2023) 195–225. doi:10.1017/S1471068421000478.

[27] V. Mellarkod, M. Gelfond, Y. Zhang, Integrating answer set programming and constraint logic programming, Annals of Mathematics and Artificial Intelligence 53 (2008) 251–287. doi:10.1007/S10472-009-9116-Y.

[28] E. Teppan, G. Friedrich, Heuristic constraint answer set programming, in: G. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, F. van Harmelen (Eds.), Proceedings of the Twenty-second European Conference on Artificial Intelligence (ECAI'16), IOS Press, 2016, pp. 1692–1693. doi:10.3233/978-1-61499-672-9-1692.

[29] S. Baselice, P. Bonatti, M. Gelfond, Towards an integration of answer set and constraint solving, in: M. Gabbrielli, G. Gupta (Eds.), Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05), volume 3668 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, pp. 52–66. doi:10.1007/11562931_7.

[30] M. Gebser, M. Ostrowski, T. Schaub, Constraint answer set solving, in: P. Hill, D. Warren (Eds.), Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), volume 5649 of *Lecture Notes in Computer Science*, Springer-Verlag, 2009, pp. 235–249. doi:10.1007/978-3-642-02846-5_22.

[31] M. Ostrowski, T. Schaub, ASP modulo CSP: The clingcon system, Theory and Practice of Logic

Programming 12 (2012) 485–503. doi:10.1017/S1471068412000142.
[32] A. De Rosis, T. Eiter, C. Redl, F. Ricca, Constraint answer set programming based on HEX-programs, in: D. Inclezan, M. Maratea (Eds.), Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'15), 2015.
[33] M. Balduccini, Y. Lierler, Constraint answer set solver EZCSP and why integration schemas matter, Theory and Practice of Logic Programming 17 (2017) 462–515. doi:10.1017/S1471068417000102.
[34] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, Satisfiability modulo theories, in: [53], 2021, pp. 1267–1329. doi:10.3233/FAIA201017.
[35] M. Bartholomew, J. Lee, First-order stable model semantics with intensional functions, Artificial Intelligence 273 (2019) 56–93. doi:10.1016/J.ARTINT.2019.01.001.
[36] T. Janhunen, G. Liu, I. Niemelä, Tight integration of non-ground answer set programming and satisfiability modulo theories, in: P. Cabalar, D. Mitchell, D. Pearce, E. Ternovska (Eds.), Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11), 2011, pp. 1–13.
[37] D. Shen, Y. Lierler, SMT-based constraint answer set solver EZSMT+ for non-tight programs, in: M. Thielscher, F. Toni, F. Wolter (Eds.), Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'18), AAAI Press, 2018, pp. 67–71.
[38] G. Liu, T. Janhunen, I. Niemelä, Answer set programming via mixed integer programming, in: G. Brewka, T. Eiter, S. McIlraith (Eds.), Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12), AAAI Press, 2012, pp. 32–42.
[39] V. Mellarkod, M. Gelfond, Integrating answer set reasoning with constraint solving techniques, in: J. Garrigue, M. Hermenegildo (Eds.), Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08), volume 4989 of *Lecture Notes in Computer Science*, Springer-Verlag, 2008, pp. 15–31. doi:10.1007/978-3-540-78969-7_4.
[40] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, Clingcon: The next generation, Theory and Practice of Logic Programming 17 (2017) 408–461. doi:10.1017/S1471068417000138.
[41] B. De Cat, B. Bogaerts, J. Devriendt, M. Denecker, Model expansion in the presence of function symbols using constraint programming, in: A. Brodsky (Ed.), Proceedings of the Twenty-fifth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'13), IEEE Computer Society Press, 2013, pp. 1068–1075. doi:10.1109/ICTAI.2013.159.
[42] C. Drescher, T. Walsh, Answer set solving with lazy nogood generation, in: A. Dovier, V. Santos Costa (Eds.), Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12), Leibniz International Proceedings in Informatics (LIPIcs), 2012, pp. 188–200. doi:10.4230/LIPICS.ICLP.2012.188.
[43] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.
[44] O. Ohrimenko, P. Stuckey, M. Codish, Propagation via lazy clause generation, Constraints 14 (2009) 357–391. doi:10.1007/S10601-008-9064-X.
[45] L. Perron, V. Furnon, OR-Tools, 2026. URL: https://developers.google.com/optimization/.
[46] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: M. Carro, A. King, N. Saeedloei, M. De Vos (Eds.), Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16), volume 52 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:15. doi:10.4230/OASICS.ICLP.2016.2.
[47] M. M. S. El-Kholany, M. Gebser, K. Schekotihin, Decomposition strategies and multi-shot ASP solving for job-shop scheduling, Logical Methods in Computer Science 21 (2025). doi:10.46298/LMCS-21(3:16)2025.
[48] R. Kolisch, A. Sprecher, PSPLIB – A project scheduling problem library: OR Software – ORSEP Operations Research Software Exchange Program, European Journal of Operational Research 96 (1997) 205–216. doi:10.1016/S0377-2217(96)00170-1.
[49] R. Kaminski, T. Schaub, On the foundations of grounding in answer set programming, Theory

and Practice of Logic Programming 23 (2023) 1138–1197. doi:10.1017/S1471068422000308.
[50] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, T. Schaub, Abstract gringo, Theory and Practice of Logic Programming 15 (2015) 449–463. doi:10.1017/S1471068415000150.
[51] S. Cotton, O. Maler, Fast and flexible difference constraint propagation for DPLL(T), in: A. Biere, C. Gomes (Eds.), Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06), volume 4121 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006, pp. 170–183. doi:10.1007/11814948_19.
[52] R. Kolisch, R. Padman, An integrated survey of deterministic project scheduling, Omega 29 (2001) 249–272. doi:10.1016/S0305-0483(00)00046-3.
[53] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2021. doi:10.3233/FAIA336.
[54] K. Baker, Introduction to Sequencing and Scheduling, John Wiley & Sons, 1974.
[55] E. Taillard, Benchmarks for basic scheduling problems, European Journal of Operational Research 64 (1993) 278–285. doi:10.1016/0377-2217(93)90182-M.
[56] M. Alviano, C. Dodaro, J. Marques-Silva, F. Ricca, Optimum stable model search: Algorithms and implementation, Journal of Logic and Computation 30 (2020) 863–897. doi:10.1093/LOGCOM/EXV061.
[57] S. Hölldobler, N. Manthey, P. Steinke, A compact encoding of pseudo-Boolean constraints into SAT, in: B. Glimm, A. Krüger (Eds.), Proceedings of the Thirty-fifth Annual German Conference on Artificial Intelligence (KI'12), volume 7526 of *Lecture Notes in Computer Science*, Springer-Verlag, 2012, pp. 107–118. doi:10.1007/978-3-642-33347-7_10.
[58] C. Sinz, Towards an optimal CNF encoding of Boolean cardinality constraints, in: P. van Beek (Ed.), Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05), volume 3709 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, pp. 827–831. doi:10.1007/11564751_73.
[59] L. Lan, J. Berkhout, PyJobShop: Solving scheduling problems with constraint programming in Python, CoRR abs/2502.13483 (2025). doi:10.48550/arXiv.2502.13483.