Revisiting Optional Variables in Lazy Clause Generation Solvers for Flexible Scheduling

Arthur Bit-Monnot
LAAS-CNRS, Université de Toulouse, INSA, CNRS, France

Abstract

Optional variables have for a long time been identified as a crucial component of state-of-the-art scheduling solvers. They not only offer a natural way to model many scheduling problems but also play an important role in the efficiency of such solvers by enabling a stronger inference in a constraint propagation engine. Despite this recognition, their adoption in modern solvers remains extremely scarce presumably because their native support requires pervasive changes to a solver and handling many subtleties that have not been extensively studied.

In this paper, we aim to provide a good foundation in this direction by providing a formal characterization of optional variables and of their interactions with key components of a constraint programming solver including propagators, reification and lazy clause generation. In addition, we present a new lazy clause generation solver with native support for optional variables and demonstrate its efficiency on variants of the flexible jobshop problem.

2012 ACM Subject Classification Computing methodologies → Discrete space search

Keywords and phrases Constraint Programming, Scheduling, Lazy-Clause Generation, Optional Variables

Digital Object Identifier 10.4230/LIPIcs.CP.2026.7

Supplementary Material *Software (Source Code)*: `https://github.com/plaans/aries`

Funding This research was partly funded by the Agence Nationale de la Recherche (ANR) under project HumFleet (ANR-23-CE33-0003-01) and benefited from the AI Interdisciplinary Institute ANITI, funded by the France 2030 program under the Grant agreement n°ANR-23-IACL-0002.

1 Introduction

In scheduling problems, it is often the case that some activities are optional, in which case the scheduling solver must not only decide when to execute activities but also which ones. Another common case is the presence of alternative ways to execute a task, which can be naturally modeled as a set of optional tasks, each with its specificity, among which the scheduling solver is required to select exactly one. The ubiquity of this pattern in scheduling very early motivated first-class support in scheduling models and solvers [1]. Since this initial work, the proposal of conditional time-intervals as a first-class modeling concept in CPOptimizer has streamlined the concept [11, 12]. Conditional time-intervals allow representing, as a single composite variable, an activity whose start time, duration, end time, and, importantly for us, presence may be decided by the solver. The benefits are two-fold. First, conditional intervals provide important modeling capabilities with many high-level constraints that capture common requirements in scheduling problems [14]. Second, their first-class support in a solver allows a limited but efficient form of conditional reasoning, allowing the solver to track facts of the form, e.g., "*if the activity is present, then it must start no later than 10*". When supported in constraint propagation, such facts can lead to further inference, strengthening the propagation process. Besides CPOptimizer, which pioneered and streamlined the approach, optional intervals have been more recently proposed as a central concept in OptalCP, another recent commercial CP solver focused on scheduling [27, 10].

Besides these solvers, the notion of conditional task intervals has also been exploited for modeling using a boolean variable to encode an interval's presence and specialized constraints to act upon them [28, 26]. In the absence of first-class optional intervals, the conditional reasoning remains limited to the propagator and cannot be shared among constraints. Important work in this context is that of Schutt et al. [23], which *(i)* extends the concept of optional intervals to general optional variables, *(ii)* provides an encoding that retains the conditional reasoning capabilities in a traditional CP solver, and *(iii)* demonstrates its integration in a lazy clause generation (LCG) solver. This combination of optional variables and lazy clause generation was demonstrated to be very efficient on flexible jobshop problems. On the modeling side, the support for optional variables was brought to the MiniZinc modeling language a year later [17]. The authors present the many possible interpretations of optional variables in various contexts, their semantics in MiniZinc, as well as the procedure to compile them away. While the authors left open the possibility of having a solver natively support optional variables, to the best of our knowledge, there is no solver that does.

Despite their obvious desirable properties, as demonstrated by state-of-the-art scheduling solvers, optional variables have not emerged as a central component in recent non-commercial solvers. One reason for this may be that optional variables carry very subtle semantics that have not been well characterized, as most of the previous work focused on a very pragmatic and case-by-case analysis of their integration into specific constraints. While the integration of native support for optional variables in a solver is already a pervasive task, this lack of a clear model complicates the exercise even more. In this paper, we wish to advance this state of affairs by providing a principled integration into a modern constraint programming solver. In particular, we build our formalization from the ground up, starting from optional variables themselves towards a characterization of their impact on constraint propagation, reification, and lazy clause generation. Based on those principles, we present a new LCG constraint programming solver with native support for optional variables and show it to be competitive with the state-of-the-art commercial solvers in flexible scheduling problems.

2 Solver Fundamentals

2.1 Background: Non-Optional Variables and Domains

We use as a basis the formalism of Aries [4], a recent LCG solver with a focus on scheduling. The central component in Aries and most finite domain solvers is the set of integer *variables* $\mathcal{X}$ and the set of domains $\mathcal{D}$.

▶ **Definition 1** (Domains). *A set of domains $\mathcal{D}$ defines an upper bound $ub_{\mathcal{D}}(x) \in \mathbb{Z}$ and a lower bound $lb_{\mathcal{D}}(x) \in \mathbb{Z}$ for each variable $x \in \mathcal{X}$.*

The domain of a variable x is the set of values it may assume and noted $\mathcal{D}(x) = [lb_{\mathcal{D}}(x) \,..\, ub_{\mathcal{D}}(x)]$. Variables are typically given an initial domain $\mathcal{D}^0$, which is iteratively restricted into an *assignment*, where a single value persists in the domain of each variable.

▶ **Definition 2** (Assignment). *An* assignment *θ is a special case of a set of domains where each variable in $\mathcal{X}$ has exactly one value in its domain.*

When considering non-optional variables only, this is equivalent to stating that, for each variable $x \in \mathcal{X}, lb_\theta(x) = ub_\theta(x)$.

A *literal* is a boolean statement on one of the bounds of a variable of the form $[\![x \leq v]\!]$ or $[\![x \geq v]\!]$, for $x \in \mathcal{X}$ and $v \in \mathbb{Z}$. A literal $[\![x \leq v]\!]$ is said to be *entailed* by a domain $\mathcal{D}$ iff $ub_{\mathcal{D}}(x) \leq v$. The negation of a literal $[\![x \leq v]\!]$ is the literal $[\![x > v]\!]$ and vice versa. We denote as $\mathcal{L}_{\mathcal{X}}$ the set of all bound literals over variables $\mathcal{X}$.

We consider a special *zero variable* ζ, whose domain only contains the value 0. The tautological literal $\top$ is defined as a bound literal on the zero variable $\top \doteq [\![\zeta \leq 0]\!]$ and the *absurd literal* is defined as its negation: $\bot \doteq \neg\top = [\![\zeta \geq 1]\!]$.

As in ARIES, we often rely on signed variables $\mathcal{Y}$ to streamline the formalization, where a signed variable $y \in \mathcal{Y}$ is either x or $-x$ for any $x \in \mathcal{X}$. Any literal in $\mathcal{L}_{\mathcal{X}}$ can be rewritten into a canonical form $[\![y \leq v]\!]$, with $y \in \mathcal{Y}$ and $v \in \mathbb{Z}$.

2.2 Optional Variables

In a regular CSP, all variables must be assigned an integer value in the solution. Optional variables differ in that they may be explicitly marked as absent. An optional variable can be seen as a variable with an extended domain $[lb_{\mathcal{D}^0}(x) \,..\, ub_{\mathcal{D}^0}(x)] \cup \{\circledast\}$ where $\circledast$ is a special value representing the absence of a variable.[1]

In order to represent optional variables, we require that each variable $x \in \mathcal{X}$ be associated with a *presence literal* $\widehat{x} \in \mathcal{L}_{\mathcal{X}}$. We say that a variable x is *mandatory* (or non-optional) if its presence literal is the tautological literal ($\widehat{x} = \top$) and *optional* otherwise. We impose two restrictions on presence literals:

1. If a variable $x \in \mathcal{X}$ appears in a presence literal, then it must be mandatory ($\widehat{x} = \top$).
2. The presence literal of a variable x may not refer to x itself, i.e., it may not have a presence literal of the form $[\![\pm x \leq v]\!]$. The only exception is the zero-variable ($\widehat{\zeta} = \top = [\![\zeta \leq 0]\!]$).

Definition 1 that defines a set of domains as a set of upper and lower bounds remains sufficient for our needs. We should however redefine the domain associated to a variable:

$$\mathcal{D}(x) = \begin{cases} [lb_{\mathcal{D}}(x) \,..\, ub_{\mathcal{D}}(x)] & \text{if } \mathcal{D} \models \widehat{x} \\ \{\circledast\} & \text{if } \mathcal{D} \models \neg\widehat{x} \\ [lb_{\mathcal{D}}(x) \,..\, ub_{\mathcal{D}}(x)] \cup \{\circledast\} & \text{otherwise} \end{cases} \tag{1}$$

That is, the variable is *absent* ($\circledast$) if its presence literal is refuted, integral if its presence literal is entailed. When neither holds (the presence literal is undetermined), the variable may be either present or absent. When a variable is mandatory, its presence $\top$ is necessarily entailed by $\mathcal{D}$ and the formulation is equivalent to the previous one.

An *assignment* remains defined as per Definition 2, that is, an assignment θ is a set of domains, defined as integral bounds, such that all variables in $\mathcal{X}$ have only one possible value. In the presence of optional variables, this condition is verified iff $\forall x \in \mathcal{X}, \theta \models \neg\widehat{x} \vee lb_\theta(x) = ub_\theta(x)$. This formalizes the intuition that, if an optional variable is absent, then its integral bounds do not play any role in defining its value and are not required to meet.

▶ **Example 3.** Consider a problem with variables $\mathcal{X} = \{p_1, s_1, d_1, p_2, s_2, d_2\}$, where, for $i \in \{1, 2\}$, $\mathcal{D}^0(p_i) = [0 \,..\, 1]$, $\widehat{p_i} = \top$, $\widehat{s_i} = \widehat{d_i} = [\![p_i \geq 1]\!]$.

[1] Previous work [11, 23] has used the bottom symbol $\bot$ to represent the absence of a variable, presumably due to its relationship with the bottom type in type theory. While our interpretation is the same, we do not adopt their notation to avoid conflicting with its usage for contradiction and falsity in boolean logic.

These variables provide a natural way to represent optional tasks (s_i, d_i), where s_i is the optional variable representing the start time, d_i the optional variable representing the duration, and p_i the binary mandatory variable representing the presence of the task in the solution and thus associated as a presence literal to both s_i and d_i.

Possible assignments for these variables would be, for instance:

1. $\{ p_1 = 0, p_2 = 0 \}$, where neither task is present and $s_1 = d_1 = s_2 = d_2 = \circledast$.
2. $\{ p_1 = 1, s_1 = 4, d_1 = 3, p_2 = 0 \}$ where only the first task is present, starts at 4 and lasts 3 time units.
3. $\{ p_1 = 1, s_1 = 4, d_1 = 3, p_2 = 1, s_2 = 10, d_2 = 3 \}$, where both tasks are present.

2.3 Domain Operations

First and foremost, it should be noted that our definition of entailment remains the same as for non-optional variables: $\mathcal{D} \models [\![y \leq v]\!] \Leftrightarrow ub_{\mathcal{D}}(y) \leq v$. A critical observation is that the entailment of $[\![y \leq v]\!]$ *does not* imply that y will have an integral value of at most v but only that *if y is present, it will have an integral value of at most v*. I.e., it does not forbid the $\circledast$ assignment.

Let us now turn our attention to how domains can be updated. There are a few desirable properties that we would like to see respected:

1. It should never be the case that a variable has an empty domain. This corresponds to a failure, as the domains could never be transformed into an assignment.
2. It should never be the case that, for a variable $x \in \mathcal{X}$, $lb_{\mathcal{D}}(x) > ub_{\mathcal{D}}(x)$. From our previous definitions, this would be valid if x is absent, but it is a general assumption in propagators (not necessarily designed with optional variables in mind) that the integral domains are non-empty for a consistent model.
3. Updating a bound of a variable already determined to be absent should have no effect. Its integral bounds play no role in the solution and any inference on them is superfluous.

These principles are implemented by the $\textsc{SetUB}(y, v)$ operation, that, given a signed variable $y \in \mathcal{Y}$, enforces it to be either absent or have an integral value that is at most $v \in \mathbb{Z}$. Hence, it either updates the integral domain or marks the variable as absent if it would result in an empty integral domain. It returns a failure when the domain of a mandatory variable could not be updated. Note that we restricted presence literals to be on mandatory variables only, so this will occur after at most one recursion. The definitions short-circuit into a no-op if the updated variable is already absent. The two other operations of interest, setting a lower bound $\textsc{SetLB}(y, v)$ and enforcing a literal $\textsc{Set}(l)$, are simply syntactic sugar.

$$\textsc{SetUB}(y, v) : \begin{cases} \text{no-op} & \text{if } \mathcal{D} \models \neg\widehat{y} \vee ub_{\mathcal{D}}(y) \leq v \\ \mathcal{D}(y) := [lb_{\mathcal{D}}(y) \mathinner{..} v] & \text{elif } lb_{\mathcal{D}}(y) \leq v \\ \textsc{Fail} & \text{elif } \widehat{y} = \top \ (y \text{ is mandatory}) \\ \textsc{Set}(\neg\widehat{y}) & \text{otherwise} \end{cases} \tag{2}$$

$$\textsc{SetLB}(y, v) : \textsc{SetUB}(-y, -v) \tag{3}$$

$$\textsc{Set}([\![y \leq v]\!]) : \textsc{SetUB}(y, v) \tag{4}$$

2.4 Compound Expressions and Constraints

So far we have only considered variables, domains, and assignments, but the other crucial component of constraint programming is constraints, that is, boolean expressions that should be entailed by any valid assignment.

2.4.1 Scopes

For our tiny scheduling problem (Example 3), a natural constraint to state would be $s_1 + d_1 \leq s_2$, i.e., the first task should finish before the second starts. This requirement is obviously satisfied for the third assignment, however it is ambiguous for the first two where at least s_2 is absent, and thus cannot be evaluated to any meaningful integer value. In a scheduling context, the most natural interpretation would be to interpret the constraint as "*if both tasks are present, then the first one should end before the second starts*" (which corresponds to the *end-before-start* constraint in CPOptimizer [13]).

This is far from the only interpretation however, even for apparently similar expressions. Assume now that each task is associated to a cost variable c_i whose presence is also tied to the presence of the task ($\widehat{c_i} = [\![p_i \geq 1]\!]$). It would be natural to bound the total cost with a constraint $c_1 + c_2 \leq 12$. Despite having the same structure (a linear constraint over optional variables), it does not make sense to consider the constraint as satisfied as soon as one task is absent: the remaining one should still fulfill the maximum cost requirement. Here a more sensible route would be to interpret an absent value as not taking part in the sum expression.

The multiple interpretations of seemingly simple expressions in the presence of optional variables and undefined values has been recognized for a long time, and notably explored in depth in the introduction of option types in MiniZinc [17]. In particular, the authors identify two main interpretations. The *projection interpretation* corresponds to our precedence example where a constraint is relaxed when some of its elements are absent. The *compression interpretation* captures our cost example where elements of a set are ignored when absent. Rather than enumerating the possible interpretations of all constraints, we aim to define the common principles that would allow arbitrary constraints to be integrated in the solver.

We define the *scope* of a constraint as the subset of the solution space where a constraint is defined. More formally, the scope $scope(c)$ of constraint c is a conjunction of literals such that, when all such literals are entailed, the constraint is defined.

▶ **Definition 4** (Scope). *The scope of a constraint c, noted $scope(c)$, is a conjunction of literals such that, for any assignment θ for which $\theta \models scope(c)$, the constraint c can be determined to be either satisfied or violated ($\theta \models c \vee \theta \models \neg c$).*

▶ **Example 5.** Consider the precedence constraint of Example 3. In a traditional CP solver such as Gecode [22] or ORTools CpSat [20], without optional variables, this could be implemented with a regular linear constraint of the form $\texttt{SumLeq}(\{ s_i, d_i, -s_2 \}, 0)$. Without a native understanding of optional variables, the constraint should be inhibited until both tasks are determined to be present. In, e.g., ORTools CpSat, this could be captured by a conditional activation of the constraint (with the combinator $\texttt{OnlyEnforceIf}(p_1, p_2)$). In our context, the scope of the constraint would similarly capture this information that the constraint is defined and should be enforced when both tasks are present: $scope(\texttt{SumLeq}(\{ s_i, d_i, -s_2 \}, 0)) = [\![p_1 \geq 1]\!] \wedge [\![p_2 \geq 1]\!]$.

On the other hand, scheduling solvers with native support for optional intervals such as CPOptimizer [13] and OptalCP [27] provide specialized constraints for this case such as $\texttt{EndBeforeStart}((s_1, d_1, p_1), (s_2, d_2, p_2))$ where each (s_i, d_i, p_i) represents an optional task with presence p_i, starting at s_i with a duration d_i. The implementation already accounts for optional variables and can be always activated (the decision of when to propagate is delegated to the constraint itself).

2.4.2 Half-Reified Constraints

We adopt a unifying view of constraints as *half-reified* constraints [7]. In the definition of Feydy et al. [7], a half-reified constraint is of the form $\ell \longrightarrow c$ (read "ℓ *half-reifies* c") where ℓ is a boolean variable (a literal in our formalism) and c is a regular constraint. It should be interpreted as "*when* ℓ *is true then* c *must be satisfied*", which directly mirrors the logical implication suggested by the notation.

Half-reified constraints are particularly interesting in that they allow capturing three common cases of constraint-based modeling:

- Regular constraint application can be modeled as $\top \longrightarrow c$ and requires the constraint c to be always active.
- Conditional constraint application, where a constraint c is only required to be active when a condition ℓ is satisfied.
- Full reification, $\ell \longleftrightarrow c$, is equivalent to two half-reifications $\ell \longrightarrow c$ and $\neg\ell \longrightarrow \neg c$, without loss of propagation strength [7].

An immediate usage of half-reified constraints is to use the scope to make the application of the constraint conditional: $scope(c) \longrightarrow c$. While this generalizes a common modeling practice for conditional constraints, it does not capture the original semantics of reification.

Let us now take a step back to consider the implications of having optional variables, where constraints may have a limited scope. The general understanding and usage of (full) reification is that, given a constraint c, one wants a boolean variable ℓ that is true iff c is satisfied. This process allows using a constraint as a boolean expression in other constraints, by replacing it with an equivalent boolean variable through a *flattening* process and is essential in compiling high-level modeling languages such as MiniZinc [19].

In the presence of optional variables, this is however limiting: outside of its scope, one may not be able to determine whether a constraint is satisfied or violated. Let us propose an initial definition stating the minimal requirements for half-reification.

▶ **Definition 6** ((Weak) Half-Reification). *A half-reified constraint $\ell \longrightarrow c$ for a literal ℓ and a constraint c requires that if ℓ is present ($\mathcal{D} \models \widehat{\ell}$) and entailed ($\mathcal{D} \models \ell$), then the constraint c must be satisfied ($\mathcal{D} \models c$).*

Let us now explore the consequences of this minimal definition. Table 1 introduces the truth table under the (reasonable) assumption that $\ell \longleftrightarrow c \Leftrightarrow \ell \longrightarrow c \wedge \neg\ell \longrightarrow \neg c$. Observe that when the value of c cannot be determined (it is outside of its scope), being consistent with a full reification requires ℓ to be absent. Stated formally, it means that $(\ell \longleftrightarrow c) \Longrightarrow (\widehat{\ell} \Rightarrow scope(c))$. While not a direct requirement of Definition 6, we propose to extend it to all half-reified constraints to maintain the alignment with full reification.

▶ **Definition 7** (Strong Half-Reification). *A strongly half-reified constraint is a half-reified constraint as per Definition 6 for which it is the case $\widehat{\ell} \Longrightarrow scope(c)$.*

▶ **Example 8.** Suppose one wants to reify a constraint $x = 6$ for some optional variable $x \in \mathcal{X}$. If x is absent, then the constraint is undefined (cannot be evaluated) and one would expect its reification variable to be absent as well. This can be captured by introducing a new binary variable $x_{=6}$ with presence literal $\widehat{x}$ (the scope of the constraint) and posting two half-reification constraint $[\![x_{=6} \geq 1]\!] \longrightarrow (x = 6)$ and $[\![x_{=6} < 1]\!] \longrightarrow (x \neq 6)$. Notice that the two half-reifications are strong, and that it is in fact the only possibility to have a single reification literal capture all of the positive, negative and undefined cases.

Table 1 Truth table for half and full reification according to Definition 6. We use the symbol $\circledast$ to denote the absence of ℓ and the symbol ? to denote that c cannot be determined to be satisfied or violated (i.e., it is outside of its scope).

ℓ	c	$\ell \longrightarrow c$	$\neg\ell \longrightarrow \neg c$	$\ell \longleftrightarrow c$
$\top$	$\top$	✓	✓	✓
$\top$	$\bot$	✗	✓	✗
$\top$	?	✗	✓	✗
$\bot$	$\top$	✓	✗	✗
$\bot$	$\bot$	✓	✓	✓
$\bot$	?	✓	✗	✗
$\circledast$	$\top$	✓	✓	✓
$\circledast$	$\bot$	✓	✓	✓
$\circledast$	?	✓	✓	✓

In the remainder of this paper, we assume that all constraints are *strongly* half-reified. This serves an important uniformization objective, with the scope of any constraint being entailed by the presence of the reifying literal which simplifies propagation rules in the presence of optional variables.

Another important consequence of our definitions is that, even in the case of full reification $\ell \longleftrightarrow c$, it is valid to have ℓ absent and c satisfied, as it can be seen in Table 1. We would argue that this is in fact a desirable for a constraint solver. Consider two optional variables x and y and a constraint $c \doteq (x \leq y)$, with $scope(c) = \widehat{x} \wedge \widehat{y}$. The constraint c can be declared *satisfied* for any domains $\mathcal{D}$ such that $ub_{\mathcal{D}}(x) \leq lb_{\mathcal{D}}(y)$, even if the presence of x and y is yet to be determined. Definition 7 ensures that the satisfaction status of c remains irrelevant whenever c is out of scope.

3 Conflicts Set and Propagation

3.1 Constraints as Conflict Sets

The general understanding of constraints is that they define a set of allowed assignments of variables, either extensively or implicitly. Given an assignment θ and a constraint c, we write $\theta \models c$, if θ is part of the allowed assignments of c. Given a set of constraints C, a *valid* assignment is any assignment θ compatible with the initial domains such that $\forall c \in C, \theta \models c$.

In our context, we instead propose to reason on the assignments forbidden by a constraint.

▶ **Definition 9** (Conflict Set). *A* conflict set *is a set of literals that cannot hold in any valid assignment of the variables. Given a set of literals* $\{\ell_1, \ell_2, \dots \ell_n\}$*, we denote the corresponding conflict set as* $\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n \rightarrow \bot$

We say that an assignment θ is *rejected* by a conflict set γ iff $\forall \ell \in \gamma, \theta \models \ell$, i.e., all literals in the conflict set are entailed, leading to a contradiction.

The presence of optional variables however poses a challenge in this interpretation. Consider the conflict set with a sole literal $[\![x \leq 7]\!]$, it is unclear whether the literal would be entailed by an assignment θ where $\theta(x) = \circledast$ (it may be entailed or negated in different domains leading to this assignment). To ease the formalization, we require that a conflict set decides the presence of any optional literal it refers to.

▶ **Definition 10** (Well-formed Conflict Set). *We say that a conflict set* γ *is well-formed, if for any optional literal* $\ell \in \gamma$*, there is a mandatory literal* $\ell' \in \gamma$ *such that* $\ell' \rightarrow \widehat{\ell}$*. We say that the literal* ℓ *is* guarded *by* ℓ'*.*

We denote as Γ_c the set of all conflict sets associated to a constraint c which such that, for any assignment θ, $\theta \models c \iff \nexists \gamma \in \Gamma_c$ s.t. $\theta \models \gamma$. I.e., the conflict sets do not reject any valid assignment and any invalid assignment is rejected by a conflict set.

▶ **Example 11.** Given two optional variables x and y, consider the constraint c requiring that if both x and y are present, then x must be greater than or equal to y ($x \geq y$).

An example conflict set for c would be $\widehat{x} \wedge \widehat{y} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightarrow \bot$. More generally the conflict sets of c would be defined as $\Gamma_c = \bigcup_{v \in \mathbb{Z}} \{\, \widehat{x} \wedge \widehat{y} \wedge [\![x \leq v]\!] \wedge [\![y > v]\!] \rightarrow \bot \,\}$.

3.2 Propagators from Conflict Sets

A naïve and lazy propagator for a constraint c would simply reject any assignment θ such that $\exists \gamma \in \Gamma_c$ such that $\theta \models \gamma$. In constraint programming, we are typically interested in stronger propagators that are capable of inferring facts based on the current domains. We capture this capability with a simple definition of a propagator, similar to the *propagation rules* of Choi et al. [5].

▶ **Definition 12** (Propagator). *A propagator p is of the form $\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n \rightsquigarrow \ell$ where ℓ and all ℓ_i are literals in $\mathcal{L}_\mathcal{X}$.*

The literals of the left-hand side are the implicants *($impl(p) \doteq \{\, \ell_1, \ldots, \ell_n \,\}$) and the right-hand side literal ℓ is the* consequence *($csq(p) \doteq \ell$).*

A propagator p states that, if $\forall \ell_i, \mathcal{D} \models \ell_i \in impl(p)$, then $\mathcal{D} \models csq(p)$. In other words, if all implicants are entailed, the right-hand side literal can be propagated.

While this definition may seem limiting when compared to the very general and complex propagators available in constraint programming, it is useful to interpret such complex propagators as a collection of our elementary propagators. The complexity of the existing propagators comes from the fact that, for computation efficiency, the set of elementary propagators is not defined extensively but implicitly.

A propagator p is a valid propagator for a constraint c if, for any assignment θ such that $\theta \models c$, then one of the following is true: (1) $\theta \models csq(p)$, (2) $\theta \models \widehat{\neg csq(p)}$ or (3) there is an implicant ℓ_i of p such that $\theta \models \neg \ell_i$. In other words, the consequence is either entailed or absent, or the propagator would not trigger in the assignment.

We are now interested in determining whether it is possible to derive propagators from well-formed conflict sets. The obvious way is to define the unit-propagators of a conflict set: as soon as all but one literal of the conflict set are entailed, the remaining one can be propagated to false.

▶ **Definition 13.** *Given a conflict set γ, the associated unit-propagator for a literal $\ell \in \gamma$ is the propagator with implicants $\gamma \setminus \{\, \ell \,\}$ and consequence $\neg \ell$.*

▶ **Example 14.** In our example for constraint $x \geq y$, a conflict set $\widehat{x} \wedge \widehat{y} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightarrow \bot$ allows the definition of four unit-propagators:

$$\widehat{x} \wedge \widehat{y} \wedge [\![x \leq 7]\!] \rightsquigarrow [\![y \leq 7]\!] \qquad \widehat{x} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightsquigarrow \neg \widehat{y}$$
$$\widehat{x} \wedge \widehat{y} \wedge [\![y > 7]\!] \rightsquigarrow [\![x > 7]\!] \qquad \widehat{y} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightsquigarrow \neg \widehat{x}$$

An interesting consequence of this observation is the existence of a propagator for every literal in a well-formed conflict set. We are now interested in determining whether a stronger propagator can be defined in the context of optional variables.

▶ **Definition 15** (Opt-unit propagator). *Given a well-formed conflict set γ and a literal $\ell \in \gamma$, the opt-unit propagator $p_{\gamma,\ell}$ is the propagator such that:*

- $csq(p_{\gamma,\ell}) \doteq \neg\ell$
- $impl(p_{\gamma,\ell}) \doteq (\gamma \setminus \{\ell\}) \setminus \{\ell' \mid \ell' \in \gamma \wedge \widehat{\ell} \rightarrow \ell'\}$

▶ **Example 16.** Continuing our example for the constraint $x \geq y$ and a conflict set $\widehat{x} \wedge \widehat{y} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightarrow \bot$, it allows the definition of four opt-unit propagators. Compared to unit propagators, this means removing from the implicants any literal that is implied by the *presence literal* of the consequence. In our case, this removes respectively $\widehat{y}$ and $\widehat{x}$ from the implicants on the two constraints on the left.

$$\begin{array}{ll} \widehat{x} \wedge [\![x \leq 7]\!] \rightsquigarrow [\![y \leq 7]\!] & \widehat{x} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightsquigarrow \neg\widehat{y} \\ \widehat{y} \wedge [\![y > 7]\!] \rightsquigarrow [\![x > 7]\!] & \widehat{y} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!] \rightsquigarrow \neg\widehat{x} \end{array}$$

Furthermore, we observe that the two propagators on the right are redundant with the ones of the left. Indeed, if $\mathcal{D} \models \widehat{x} \wedge [\![x \leq 7]\!] \wedge [\![y > 7]\!]$, the top-left propagator, would propagate $[\![y \leq 7]\!]$ which is in contradiction with the current lower bound of 8 and $\textsc{SetUB}(y, 7)$ would impose $\neg\widehat{y}$ (Equation (2)).

▶ **Theorem 17.** *An opt-unit propagator $p_{\gamma,\ell}$ is a valid propagator for a well-formed conflict set γ.*

Proof (Sketch). One can show that the removal of an implicant either results in a propagation that is valid or a no-op in an assignment where the consequence variable is absent. The full proof is given in Appendix A ◀

In this section, we provided a characterization of constraints through conflict sets and a mechanical procedure to derive valid propagators that account for the optionality of variables, that can be stronger than the usual unit propagation. These insights may provide guidance and assurance in defining new specialized propagators, which we do in Section 5. The concepts introduced are also important to understanding the validity of lazy clause generation in the presence of optional variables (Section 4). But before that, we discuss how existing propagators that have been developed in the community for decades fit into the picture.

3.3 Reusing Existing Propagators

In this section, we assume that we are given a constraint c with an existing propagation algorithm that is not aware of optional variables. In this context, it must be the case that $scope(c) \rightarrow \widehat{x}$ for any variable x appearing in c. Assuming c is strongly half-reified by a literal ℓ ($\ell \longrightarrow c$), a first, mostly theoretical, insight is that the half-reified constraint has a set of conflict set $\{\gamma \cup \{\ell, \widehat{\ell}\} \mid \gamma \in \Gamma_c\}$, by Definition 6. By Definition 7, it follows that the resulting conflict sets are well-formed: any optional variable must be present when the variable is in scope, and hence any optional literal in the original conflict set would be guarded by $\widehat{\ell}$. Having established that, we discuss a number of rules that can be leveraged to propagate c in the presence of optional variables. These are given in Table 2, for an arbitrary half-reified constraint $\ell \longrightarrow c$.

Rules (1.) and (2.) simply derive from Definition 6 that does not requires anything when ℓ is absent or falsified. Rule (3.) is also direct from Definition 6, when ℓ is present and entailed, c must be satisfied which allows the constraint to propagate. In other words, rules (1-3) implements the left to right propagation of $\widehat{\ell} \wedge \ell \implies c$.

Table 2 Propagation rules for a half-reified constraint $\ell \longrightarrow c$.

Rule	Condition	Do
1.	$\mathcal{D} \models \neg\ell$	Nothing
2.	$\mathcal{D} \models \neg\widehat{\ell}$	Nothing
3.	$\mathcal{D} \models \widehat{\ell} \wedge \ell$	Normal propagation of c
4.	$\mathcal{D} \models \neg c$	$\textsc{Set}(\neg\ell)$
5.	$\mathcal{D} \models \ell$	Propagation restricted to variables $x \in c$ such that $\widehat{x} \rightarrow \widehat{\ell}$

Rule (4.) is a form of eager propagation for the converse. Whenever c cannot be satisfied with the current domains, it must be the case that $\neg\widehat{\ell} \vee \neg\ell$ holds. This is precisely the semantics of $\textsc{Set}(\neg\ell)$ (from Equation (4)): it enforces ℓ to be either false or absent (i.e., it removes the truth value from the domain). Propagators can support this rule by having a satisfiability check as done in regular half-reification [7].

Rule (5.) is of course the most interesting one to enable eager propagation on optional variables. To understand how this is made possible by optional variables, assume that the propagator has restricted the bound of a variable $x \in c$, such that $\widehat{x} \rightarrow \widehat{\ell}$. We of course assume that this propagation is valid in normal propagation (under rule 3). Rule 5 is only applicable when $\widehat{\ell}$ is yet undetermined ($\mathcal{D} \not\models \widehat{\ell}$ and $\mathcal{D} \not\models \neg\widehat{\ell}$). Its value must however be set in any assignment θ constructed from $\mathcal{D}$. Assume $\theta \models \widehat{\ell}$, the inference made would be propagated by rule 3, and was thus a correct, eager, inference. Assume $\theta \models \neg\widehat{\ell}$, then $\theta \models \neg\widehat{x}$ (by our enabling condition) and x is absent from the solution ($\theta(x) = \circledast$). Thus, the change to the integral domain of x that was performed is useless but correct as it does not affect the resulting assignment.

▶ **Example 18.** Consider two optional variables x and y, and a constraint $c \doteq (x \geq y)$ with scope $\widehat{x} \wedge \widehat{y}$. In our normalized form, this means the constraint is posted as a half-reified constraint $\ell \longrightarrow c$ where ℓ is such that $\widehat{\ell} \longleftrightarrow \widehat{x} \wedge \widehat{y}$ and ℓ is entailed (i.e. it is absent if x or y are and true otherwise).

If $\mathcal{D} \models \widehat{\ell} \wedge \ell \wedge [\![x \leq 7]\!]$, then the propagator could be activated (rule 3) and infer $[\![y \leq 7]\!]$. This is the normal case where the constraint is in scope and both x and y are present.

Consider now the case where $\mathcal{D} \models \widehat{x} \wedge \ell \wedge [\![x \leq 7]\!]$, a weaker case where the presence of y is not yet determined. We would like to determine whether the upper bound of y can be tightened nevertheless. Observe that $\widehat{x} \wedge \widehat{y} \rightarrow \widehat{\ell}$ by definition. Since $\mathcal{D} \models \widehat{x}$, we can infer that $\widehat{y} \rightarrow \widehat{\ell}$ in all subsequent assignments. Hence, by application of rule 5, the upper bound of y can be tightened by $\textsc{SetUB}(y, 7)$.

The strength of optional variables lies in specialized propagators that exploit this fifth rule, many of which are proposed in scheduling [1, 11, 12, 23]. We present few such propagators in Section 5, either novel or adapted from the literature.

4 Lazy Clause Generation

Before looking into specific propagators that leverage optional variables, we must address the issue of lazy clause generation, a fundamental building block of modern CP solvers [8, 21, 4]. We consider the classical setting of *conflict-driven clause learning* (CDCL) [16] used both in SAT and lazy clause generation solvers. When facing a propagation failure, such solvers would iteratively build a clause that is implied by other constraints in the problem. For this process, LCG solvers further require explaining propagators, which is immediate from our definition of a propagator: the explanation of a literal ℓ inferred by a propagator p is the set of implicants of p.

▶ **Example 19.** Continuing from Example 18, we would like the propagator of c to provide an explanation of the inferred literal $[\![y \leq 7]\!]$. In our example, a minimal explanation would be $\widehat{x} \land [\![x \leq 7]\!] \rightsquigarrow [\![y \leq 7]\!]$ where ℓ is omitted from the right-hand side because it is true in all assignments by its definition. The usual interpretation of this explanation would be as the corresponding clause $\neg\widehat{x} \lor [\![x > 7]\!] \lor [\![y \leq 7]\!]$, which is however *incorrect in the presence of optional variables*. First, it is easy to see that the clausal expression may be undefined when y is absent. Second, if we were to use this clause as a constraint, it would unit propagate $\neg\widehat{x}$ when both $[\![x \leq 7]\!]$ and $[\![y > 7]\!]$ are entailed, even though y may be absent and nothing can be inferred from its integral bounds.

The example establishes that clauses are not an appropriate model to reason about explanations in the presence of optional variables. We provide an alternative view, equivalent to the original definition, that is better adapted to our setting. Instead of a clausal representation of the implied constraint, we consider the dual view of a conflict set that does not suffer from the same problems in the presence of optional variables.

An initial conflict set originates from a propagation failure of a SetUB(x, v) call for which SetLB and Set are syntactic sugar (Equation (2)). The associated conflict set is $[\![x \leq v]\!] \land [\![x > v]\!] \land \widehat{x} \rightarrow \bot$, the only case in which a propagation would fail. Such an initial conflict set is well-formed: the bound literals of x are guarded by $\widehat{x}$.

From such a conflict set γ, CDCL refines it by replacing a literal $\ell \in \gamma$ by its implying literals, a set of literals that, when all true, imply ℓ. In the clausal interpretation, this means applying the resolution rule between the implied clause under construction and the clause that unit-propagated ℓ. In our context, and with our definition of propagators, the implying literals for ℓ are simply the implicants of the propagator that inferred p. Thus for a conflict set γ, the resolution step of a literal $\ell \in \gamma$ that was propagated by p results in a new conflict set $\gamma' = (\gamma \setminus \{\ell\}) \cup \mathit{impl}(p)$.

▶ **Lemma 20** (Well-Formedness Preservation under Resolution)**.** *Given a* well-formed *conflict set γ and a literal $\ell \in \gamma$ propagated by p, the set $\gamma' \doteq (\gamma \setminus \{\ell\}) \cup \mathit{impl}(p)$ is a well-formed conflict set.*

Proof. Assume there is an assignment θ rejected by γ' ($\theta \models \gamma'$). Given that $\mathit{impl}(p) \subseteq \gamma'$, then $\theta \models \mathit{impl}(p)$ and $\theta \models \ell$ (the propagator is assumed valid). Since $\gamma \subseteq (\gamma' \cup \{\ell\})$, it follows that $\theta \models \gamma$ meaning γ' would not reject more assignments than γ when p is propagated.

For well-formedness, we note that for a propagator p, any implicant is either guarded by another implicant or by $\widehat{\ell}$. Given that γ is well-formed, any implicant that was guarded by $\widehat{\ell}$ would be guarded by the guard of ℓ that is present in γ and would remain in γ'. ◀

▶ **Lemma 21.** *A* well-formed *conflict set $\ell_1 \land \ell_2 \land \cdots \land \ell_n \rightarrow \bot$ can be interpreted as a unit-propagable clause $\neg\ell_1 \lor \neg\ell_2 \lor \cdots \lor \neg\ell_n$.*

Proof. Direct from Theorem 17 which demonstrated the validity of the opt-unit propagator for each literal in a well-formed conflict set. As the unit propagator is less general, its existence and validity is also implied for any literal of a well-formed conflict set. ◀

We are now ready to establish our main result that CDCL resolution is applicable in the presence of optional variables.

▶ **Theorem 22.** *When given an initial, well-formed, conflict set, CDCL resolution yields a valid and well-formed conflict set whose clausal representation can be added to a clause database for unit propagation.*

Proof. The proof is immediate by recurrence: the initial conflict set being well-formed, resolution will always yield a valid and well-formed conflict set (Lemma 20), under the assumption that the propagators are valid. When the process stops, e.g., because the first unique implication (1UIP) has been reached, the resulting conflict set is thus still well-formed and can be interpreted into a unit-propagable clause (Lemma 21). ◀

5 Compound Propagators

In this section, we briefly discuss three new propagators for common constraints that can be easily derived from the introduced formalism (Sections 3.2 and 3.3).

5.1 Disjunction (clause)

A disjunction constraint (or clause in the SAT terminology) is a fundamental constraint of the form $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n$ where each ℓ_i is a literal in $\mathcal{L}_\mathcal{X}$. Note that a clause has a direct interpretation as a conflict set $\neg\ell_1 \wedge \cdots \wedge \neg\ell_n \to \bot$. Following the results of Section 3.3, we can further interpret its half reification by a literal ℓ as the conflict set $\widehat{\ell} \wedge \ell \wedge \neg\ell_1 \wedge \cdots \wedge \neg\ell_n \to \bot$, which is necessarily well-formed. From this interpretation, Theorem 17 provides us with an opt-unit propagator for each literal that may be stronger than the corresponding unit propagator.

It should be noted, however, that clauses play a very central role in modern solvers and the efficiency of their propagation is critical to performance. Detecting when a clause is opt-unit may be non-trivial and is in general not compatible with the two-watches scheme [18]. The original two-watches technique selects, for a clause with conflict set γ, a watch-set of two literals $\{a, b\} \subseteq \gamma$ such that neither are currently entailed. If this is not possible, then the clause contains either a single unrefuted literal that can be unit-propagated or none, in which case we have reached a propagation failure. The two literals are monitored such that a new two-watches set is re-selected any time one of the watches becomes entailed.

Given the conflict set γ, we consider as a propagator $\bigwedge \gamma \setminus \{\ell, \widehat{\ell}\} \rightsquigarrow \neg\ell$, a weaker form of the opt-unit propagator but stronger than the unit one. Propagation with two watches only requires the following adaptation: if the watch-set is of the form $\{\ell, \widehat{\ell}\}$, then try to select another non-entailed literal to replace $\widehat{\ell}$.[2] If that is not possible, ℓ and $\widehat{\ell}$ are the only two literals of γ that are not entailed, and we can trigger propagation of $\neg\ell$.

5.2 Difference-Logic

A difference constraint is a constraint of the form $x + \delta \leq x'$, where $\delta \in \mathbb{Z}$ and x and x' are variables in $\mathcal{X}$. In [4], it was shown that the constraint can be reformulated into a set of elementary propagators, each of the form $[\![y \leq v]\!] \rightsquigarrow [\![y' \leq v + d]\!]$ where $y \in \mathcal{Y}$ and $y' \in \mathcal{Y}$ are signed versions of x and x' and v and d are integer constants in $\mathbb{Z}$.

Let us assume that the constraint is half-reified by a literal ℓ such that $\ell \leftrightarrow \widehat{y} \wedge \widehat{y'}$, capturing the usual semantics of a precedence constraint between two optional tasks. Then rule (5.) of Table 2 states that we are allowed to trigger the propagation as soon as $\mathcal{D} \models \ell$ and that we can establish that $\widehat{y'} \to \widehat{\ell}$. This second condition is satisfied as soon as $\mathcal{D} \models \widehat{y}$ which

[2] Replacing $\widehat{\ell}$ instead of ℓ ensures that the process converges as ℓ, being optional, cannot be the presence literal of another member of the conflict set.

establishes the validity of the propagator $\ell \wedge \widehat{y} \wedge [\![y \leq v]\!] \rightsquigarrow [\![y' \leq v + d]\!]$. This propagator allows an eager propagation on the integral domain of y' as soon as the constraint is active and y is present, reproducing the result of [11].

Furthermore, rule (4.) allows us to derive the propagator $[\![y \leq v]\!] \wedge [\![y' \leq v+d]\!] \rightsquigarrow \neg\ell$ which marks the constraint as unsatisfiable as soon as the integral domains become incompatible (even if none of the considered variables are determined to be present).

These propagators are a direct generalization of the propagations rules of ARIES [4]. For the sake of space we omit the straightforward (but lengthy) adaptation of the organized propagation to these propagators.[3]

5.3 NoOverlap: Overload Checking & Edge-Finding

Let us define an optional task i as (s_i, d_i, e_i, p_i) where $s_i \in \mathcal{X}$ is the start time of i, $d_i \in \mathcal{X}$ its duration and $e_i \in \mathcal{X}$ its end time. Its presence is captured by the literal $p_i \in \mathcal{L}_\mathcal{X}$ such that $p_i = \widehat{s_i} = \widehat{d_i} = \widehat{e_i}$. Given a set $\mathcal{T}$ of optional tasks, the `NoOverlap` constraint is defined as $\texttt{NoOverlap}(\mathcal{T}) \Leftrightarrow \forall_{i \neq j \in \mathcal{T}}\ \neg p_i \vee \neg p_j \vee (e_i \leq s_j) \vee (e_j \leq s_i)$ [28].

The straightforward implementation is with half-reified difference constraints. For each distinct i and j in $\mathcal{T}$, we define a literal ℓ_{ij} such that $\widehat{\ell_{ij}} \leftrightarrow p_i \wedge p_j$ and post two half-reified propagators $\ell_{ij} \longrightarrow (e_i \leq s_j)$ and $\neg\ell_{ij} \longrightarrow (e_j \leq s_i)$. While these constraints are sufficient to capture the semantics of the `NoOverlap` constraint, we propose in addition a straightforward adaptation of the edge finding propagator to support optional tasks.

Given a set of domains $\mathcal{D}$ and a task $i \in \mathcal{T}$, we denote as est_i the *earliest start time* of i ($lb_\mathcal{D}(s_i)$), as lct_i the *latest completion time* of i ($ub_\mathcal{D}(e_i)$) and δ_i its minimal duration ($lb_\mathcal{D}(d_i)$). The notation is extended to a set of tasks $\Omega \subseteq \mathcal{T}$: $est_\Omega = \min_{i \in \Omega} est_i$, $lct_\Omega = \max_{i \in \Omega} lct_i$, and $\delta_\Omega = \Sigma_{i \in \Omega}\ \delta_i$. The *earliest completion time* of Ω is denoted as $ect_\Omega = est_\Omega + \delta_\Omega$. The literature identifies the following rules for overload checking and edge finding:

$$\text{overload:} \quad ect_\Omega > lct_\Omega \Rightarrow \bot \qquad \forall \Omega \subseteq \mathcal{T}_{mand} \tag{5}$$

$$\text{overload-opt:} \quad ect_{\Omega \cup \{\, i \,\}} > lct_{\Omega \cup \{\, i \,\}} \Rightarrow \neg p_i \qquad \forall \Omega \subseteq \mathcal{T}_{mand}, i \in \mathcal{T}_{opt} \tag{6}$$

$$\text{edge-finding:} \quad ect_{\Omega \cup \{\, i \,\}} > lct_\Omega \Rightarrow \Omega \prec i \qquad \forall \Omega \subseteq \mathcal{T}_{mand}, i \in \mathcal{T} \tag{7}$$

where $\mathcal{T}_{mand} = \{\, i \mid i \in \mathcal{T} \wedge \mathcal{D} \models p_i \,\}$ and $\mathcal{T}_{opt} = \mathcal{T} \setminus \mathcal{T}_{mand}$ respectively denote the mandatory and optional tasks. Vilim et al. [28] introduced an $O(n \times \log(n))$ algorithm for Equations (5) and (6) and for Equation (7) when i is restricted to mandatory tasks. The validity of Equation (7) when i is optional is identified as early as [1] with an $O(n^3)$ propagation algorithm.

Contrary to the observation of the authors of the $O(n \times \log(n))$ edge-finding algorithm [28], it is straightforward to allow propagation to the *bounds* of optional tasks and fully support Equation (7). In their algorithm, the authors use specific data structures to efficiently compute $\overline{ect}(\Theta, \Lambda) = \max_{i \in \Lambda} ect_{\Theta \cup \{\, i \,\}}$. The algorithm starts with $\Theta = \mathcal{T}_{mand}$ and $\Lambda = \emptyset$, and iteratively (1) transfers tasks from Θ to Λ and (2) when $\overline{ect}(\Theta, \Lambda) > lct_\Theta$, Equation (7) is triggered and the responsible i is removed from Λ. The only adaptation needed is to start with $\Lambda = \mathcal{T}_{opt}$. Any propagation remains valid by Equation (7). Importantly, it maintains the same propagation for mandatory tasks: the edge finding propagation for a task $i \in \Lambda$ cannot be blocked by another task $j \in \Lambda$ because then, edge-finding rule would also trigger for j which would be removed from Λ, unblocking the propagation of i.

[3] The only two changes are (1) to only activate an edge propagator when both ℓ and $\widehat{y}$ are true, and (2) when a cycle is detected, to make an arbitrary variable in it absent (which is what would have occurred if the propagation had continued).

The full algorithm is given in Appendix B. The presentation combines the overload checking of [28], that propagates Equation (6) and builds the data structures such that $\Theta = \mathcal{T}_{mand}$ and $\Lambda = \mathcal{T}_{opt}$, followed by the edge-finding algorithm from the same authors which, with this particular initialization, fully propagates Equation (7).

6 Experimental Results

6.1 Flexible Jobshop Scheduling Problem (FJS)

Let J be the set of jobs, M the set of machines. Each job $j \in J$ is a sequence of operations $O_{j,1}, \ldots, O_{j,n_j}$. Each operation o can be processed on a subset $M_o \subseteq M$ with processing time $p_{o,m}$ for $m \in M_o$. Each machine is constrained to process at most one operation at a time and the objective is to minimize the makespan.

For each operation o, the standard CP model uses one mandatory interval variable I_o, with start, duration and end time variables and, for each possible machine $m \in M_o$, an optional interval $I_{o,m}$ with an optional start variable and a fixed duration $p_{o,m}$. Those intervals are tied together with the following constraints:

$$\text{end}(I_{O_{j,k}}) \leq \text{start}(I_{O_{j,k+1}}) \qquad \forall j,\ k = 1, \ldots, n_j - 1 \tag{8}$$

$$\text{alternative}\big(I_o, \{\, I_{o,m} \mid m \in M_o \,\}\big) \qquad \forall o \tag{9}$$

$$\text{noOverlap}\big(\{I_{o,m} | o \text{ s.t. } m \in M_o\}\big) \qquad \forall m \in M \tag{10}$$

where Equation (8) imposes the processing order inside a job, Equation (9) imposes the choice of exactly one machine per operation and Equation (10) ensures that machines are not overloaded. The objective is to minimize the makespan $C_{\max} = \max_o \text{end}(I_o)$.

In addition to the original problem, we consider two variants: *transport-times* and *maximum time-lag* [6]. The former imposes a machine-dependent transport time between any two subsequent operations of the same job. The latter requires a maximum delay between two subsequent operations of the same job.

6.2 Compared Solvers

The approach is implemented inside the Aries solver [4]. All variables are associated with a presence literal, with $\top$ being used for mandatory variables and all constraints declare a minimal scope in which they are defined. When updating a bound of a variable (SetUB(y, v)), the domain handling logic is modified to make the variable absent if the integral domain would be empty and to generate a base conflict set $\{\, [\![y \leq v]\!], [\![y > v]\!], \widehat{y} \,\}$ which can be refined into a first UIP with the standard clause derivation procedure. Search reuses that of Aries: *Learning Rate Branching (LRB)* [15] with the exception that only variables already determined to be present are selected for branching (similar to what is done in CPOptimizer [29]).

CPOptimizer is a state-of-the-art commercial solver for scheduling [13]. It has native support for optional variables, and we use it with a single worker which exploits an automatic search with a combination of *Large Neighborhood Search* (LNS, [24]) and *Failure Directed Search* (FDS, [29]).

OptalCP is a recent commercial scheduling solver that ports most of the successful components of CPOptimizer in a modern implementation, featuring notably optional variables, LNS and an improved FDS [27, 10]. In the absence of automatic search, we refer to OptalCP$_{LNS}$ and OptalCP$_{FDS}$ as the solvers with respectively a single LNS and a single FDS worker. OptalCP refers to the solver with both workers enabled.

Table 3 Number of problems solved with and optimality proof (#Solved) and average IPC score multiplied by 100, for each of the solvers for *flexible jobshop* (FJS), *flexible jobshop with transport times* (FJS_{TT}) and *flexible jobshop with maximum time-lag* (FJS_{Lag}). The best overall result is highlighted in bold font. The best result among all solvers without LNS is underlined (among ARIES, CPSAT (1w) and OPTALCP_{FDS}).

	FJS		FJS_{TT}		FJS_{Lag}	
	#Solved	IPC (×100)	#Solved	IPC (×100)	#Solved	IPC (×100)
ARIES	261	99.45	**273**	98.75	**227**	98.89
CPOPTIMIZER	246	99.76	254	99.58	192	98.60
CPSAT (1w)	243	97.77	247	96.41	147	94.42
CPSAT (6w)	241	98.71	232	96.71	140	96.16
OPTALCP	263	99.94	265	99.83	209	**99.28**
OPTALCP_{FDS}	**265**	97.61	256	94.56	214	95.98
OPTALCP_{LNS}	108	**99.97**	50	**99.94**	42	97.82

CPSAT is a state-of-the-art CP solver bundled in Google's ORTOOLS package [20, 21] which has been the best contender in the last MINIZINC challenge [19]. Unlike CPOPTIMIZER and OPTALCP, it relies on LCG but does not support optional variables natively. CPSAT (1w) refers to the version with a single worker and CPSAT (6w) to the version with 6 workers (which was the most performant in all multi-worker configurations we tried).

ARIES uses the model of Section 6.1, implemented with the constraints of Section 5. Other solvers use the very similar model provided with the solver's distribution, with minor adaptations to support transport-times and maximum time-lag.

6.3 Experiments

We consider the 397 instances in the OPTALCP benchmark suite.[4] A description of the instances and their original sources is provided in [2]. For FJS with transport-times (FJS_{TT}), we use the layout provided in the appendix of [3]. For FJS with maximum time-lag (FJS_{Lag}), we use a maximum time lag of 10 time units between any pair of subsequent operations of the same job. All benchmarks are run on a single physical core of an AMD EPYC 9754 CPU, with a 1800 seconds timeout. As metrics, we consider the number of instances solved to optimality (#Solved) and the average IPC score [25], defined as $\frac{C^*}{C}$, where C^* is the cost of the best solution among all solvers and C is the cost of the solution (or ∞ when no solution is found).

The results for all problems are given in Table 3. A first observation is that ARIES substantially outperforms CPSAT, the other LCG solver, both in quality and number of solved instances on all problems. It also outperforms CPOPTIMIZER in solved instances on all problem classes and is only slightly behind OPTALCP in the base FJS instances.

In terms of quality, solvers with a strong LNS component (CPOPTIMIZER, OPTALCP, OPTALCP_{LNS}) tend to provide the best solutions. ARIES provides marginally longer solutions but remains by far the best among non-LNS solvers. Figure 1 provides additional insights on the dynamics over time, where LNS-based solvers very rapidly find high-quality solutions, while other solvers require more time to close the gap. Detailed results are provided in Appendix C.

[4] https://github.com/ScheduleOpt/optalcp-benchmarks/tree/main/benchmarks/flexible-jobshop

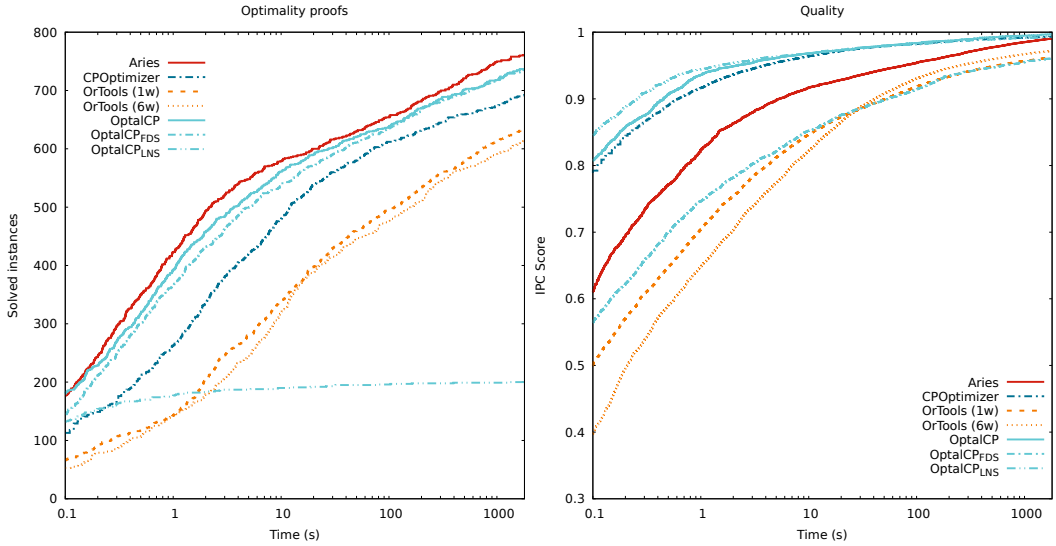

Figure 1 IPC Score and Solved instances over time for all three variants considered.

7 Conclusion

We presented a principled view of optional variables in CP, with the hope to provide a solid foundation for their integration in CP engines. We showed how these principles can be integrated into a lazy-clause generation solver, establishing minimal requirements for the compatibility of optional variables with half-reification and conflict-driven clause learning. We provided a generic framework for deriving new propagators either from first principles or from existing ones, exploited to define new propagators for clauses, half-reified difference constraints and the no-overlap constraint in the presence of optional variables.

This integration is demonstrated in Aries and tested on the flexible jobshop scheduling problem and two of its variants. Compared to commercial scheduling solvers, Aries shows very competitive performance in solution quality and in general a greater ability to prove optimality. It shows very robust performance to minor variations of the base problem, both when the delays are expanded (transport times) or restricted (maximum time-lags).

Following these results, a natural avenue for future work would be to speed up the discovery of high-quality solutions for which LNS-based solvers still display an advantage. We also wish to perform a more exhaustive characterization of the performance on additional variants of flexible scheduling problems.

References

1 J.Christopher Beck and Mark S. Fox. Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence*, 2000. doi:10.1016/S0004-3702(00)00035-7.

2 Dennis Behnke and Martin Josef Geiger. Test instances for the flexible job shop scheduling problem with work centers. Technical report, Universitätsbibliothek der HSU/UniBw H, 2012. doi:10.24405/436.

3 Lucas Berterottière, Stéphane Dauzère-Pérès, and Claude Yugma. Flexible job-shop scheduling with transportation resources. *European Journal of Operational Research*, 2024. doi:10.1016/j.ejor.2023.07.036.

4 Arthur Bit-Monnot. Enhancing Hybrid CP-SAT Search for Disjunctive Scheduling. In *European Conference on Artificial Intelligence (ECAI)*, 2023. doi:10.3233/FAIA230278.

5 C. W. Choi, J. H.M. Lee, and P. J. Stuckey. Removing propagation redundant constraints in redundant modeling. *ACM Transactions on Computational Logic*, 2007. doi:10.1145/1276920.1276925.

6 Stéphane Dauzère-Pérès, Junwen Ding, Liji Shen, and Karim Tamssaouet. The flexible job shop scheduling problem: A review. *European Journal of Operational Research*, 2024. doi:10.1016/j.ejor.2023.05.017.

7 Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half Reification and Flattening. In *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer, 2011. doi:10.1007/978-3-642-23786-7_23.

8 Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2009. doi:10.1007/978-3-642-04244-7_29.

9 Emmanuel Hebrard. Disjunctive Scheduling in Tempo. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2025. doi:10.4230/LIPIcs.CP.2025.13.

10 Vilém Heinz, Petr Vilím, and Zdeněk Hanzálek. Reinforcement learning for search tree size minimization in Constraint Programming: New results on scheduling benchmarks. *Computers & Industrial Engineering*, 2025. doi:10.1016/j.cie.2025.111413.

11 Philippe Laborie and Jerôme Rogerie. Reasoning with Conditional Time-Intervals. In *International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2008.

12 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources. In *International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2009.

13 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 2018. doi:10.1007/s10601-018-9281-x.

14 Philippe Laborie, Jérôme Rogerie, Paul Shaw, Petr Vilím, and Ferenc Katai. Interval-Based Language for Modeling Scheduling Problems: An Extension to Constraint Programming. In *Algebraic Modeling Systems*. Springer, 2012. doi:10.1007/978-3-642-23592-4_6.

15 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2016. doi:10.1007/978-3-319-40970-2_9.

16 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. doi:10.3233/FAIA200987.

17 Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace. Modelling with Option Types in MiniZinc. In *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*. Springer, 2014. doi:10.1007/978-3-319-07046-9_7.

18 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Conference on Design automation (DAC)*, 2001. doi:10.1145/378239.379017.

19 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

20 Laurent Perron and Frédéric Didier. CP-SAT, 2026. URL: https://developers.google.com/optimization/cp/cp_solver/.

21 Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP Solver (Invited Talk). In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2023. doi:10.4230/LIPIcs.CP.2023.3.

22 Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*. Gecode Project, 2019.

23 Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Scheduling Optional Tasks with Explanation. In *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer, 2013. doi:10.1007/978-3-642-40627-0_47.

24 Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 1998. doi:10.1007/3-540-49481-2_30.

25 Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, 2024. doi:10.1002/aaai.12169.

26 Petr Vilím. *Global Constraints in Scheduling*. PhD Thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, 2007. URL: https://vilim.eu/petr/disertace.pdf.

27 Petr Vilím. CP Solver Design for Maximum CPU Utilization (Invited Talk). In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2023. doi:10.4230/LIPIcs.CP.2023.5.

28 Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of O(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 2005. doi:10.1007/s10601-005-2814-0.

29 Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-Directed Search for Constraint-Based Scheduling. In *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, 2015. doi:10.1007/978-3-319-18008-3_30.

A Proofs

▶ **Theorem 17.** *An opt-unit propagator $p_{\gamma,\ell}$ is a valid propagator for a well-formed conflict set γ.*

Proof. We need to show that there is no assignment θ that is accepted by γ and where propagation would trigger and result in a contradiction.

Assume the opposite, that there exists such an assignment θ. In this setting, we know that every literal in $\mathit{impl}(p_{\gamma,\ell})$ is entailed (propagation would trigger) and at least one literal of γ is not entailed (assignment is accepted.) We need to consider three cases:

1. $\theta \models \neg\widehat{\ell}$: in this case, propagating $\neg\ell$ is a no-op and would not result in a failure (Equation (2)).
2. $\theta \models \widehat{\ell} \wedge \neg\ell$: As for the previous case, propagation is a no-op as $\neg\ell$ is already entailed.
3. $\theta \models \widehat{\ell} \wedge \ell$: here we can show that all literals of γ are entailed, which contradicts our initial assumption that the assignment is accepted by γ. We already established that literals of $\mathit{impl}(p_{\gamma,\ell})$ are entailed (propagation would trigger). For the assignment to be accepted, there should thus be at least one literal ℓ' in $(\gamma \setminus \mathit{impl}(p_{\gamma,\ell}))$ that is refuted. Those literals are the ones removed when constructing the implicants set (Definition 15). Those are either ℓ (which is entailed) or a literal ℓ' such that $\widehat{\ell} \to \ell'$. Given that $\mathcal{D} \models \widehat{\ell}$, any such ℓ' is also entailed. Hence, the assignment is rejected by γ, contradicting our initial assumption. ◀

B Detailed Edge-Finding Algorithm

Algorithm 1 provides the detailed algorithm for overload-checking and edge-finding, which is an immediate concatenation of the two algorithms of [28] from which we reuse the notation. The implementation uses the Θ-Λ-Tree of the same paper, which is shared between the overload-checking phase that builds up the tree and the edge-finding phase that empties it. Compared to the original implementation, it means that the overload-checking inside the edge-finding phase can be omitted as it was previously checked.

The propagation of the precedence $\Theta \prec i$ is done by updating the earliest starting time of i as in the original algorithm. We also considered posting the corresponding precedence constraints as done in [9] but did not find a statistically significant improvement.

Explanations are generated following the ones of Tempo [9] whose only required adaptation is the addition of the presence literals of mandatory tasks to the implicant sets.

Algorithm 1 Integrated overload-checking and edge-finding algorithm with optional variables.

// Overload checking with optional tasks
$(\Theta, \Lambda) \leftarrow (\emptyset, \emptyset)$
for $i \in T$ in non-decreasing order of lct_i **do**
  if i is optional ($\mathcal{D} \not\models \widehat{i}$) **then**
    $\Lambda := \Lambda \cup \{ i \}$
  else
    $\Theta := \Theta \cup \{ i \}$
    if $ect_\theta > lct_i$ **then**
      Fail ▷ Overload: no solution exists
  while $\overline{ect}(\Theta, \Lambda) > lct_i$ **do**
    $o :=$ optional activity responsible for $\overline{ect}(\Theta, \Lambda)$
    Set($\neg p_o$)
    $\Lambda := \Lambda \setminus \{ o \}$
// Edge-Finding propagation
$Q \leftarrow \Theta$ ▷ compulsory tasks ordered by non-increasing lct_j
$j \leftarrow peek(Q)$
while Q non-empty **do**
  $(\Theta, \Lambda) \leftarrow (\Theta \setminus \{ j \}, \Lambda \cup \{ j \})$
  $pop(Q)$
  $j \leftarrow peek(Q)$
  while $\overline{ect}(\Theta, \Lambda) > lct_j$ **do**
    $i \leftarrow$ gray activity responsible for $\overline{ect}(\Theta, \Lambda)$
    propagate: $\Theta \prec i$ ▷ $est_i \leftarrow max\{ est_i, ect_\Theta \}$
    $\Lambda \leftarrow \Lambda \setminus \{ i \}$

C Additional Results

C.1 Performance per Flexible Jobshop Variant

Figures 2, 5, and 8 provide detailed results for each of the problem variants. The associated tables provide the metrics for each of the instance sets.

A general observation is that there is always either Aries or a variant of OptalCP among the best solvers for each instance set. The general trend is that OptalCP is in general better with respect to quality while Aries is better in terms of optimality proofs.

C.2 Comparison with [23]

The approach of [23] is a very relevant work against which a comparison would have been very meaningful, as it features an LCG solver with an emulation of optional variables. We cannot provide it at this point as we did not get access to the underlying solver or detailed results. A comparison with the results presented in the paper, however, suggests that it would not be competitive with either Aries or OptalCP. For instance, their solver requires 734 seconds and a tight initial upper bound to prove optimality of all `CB` (Barnes) instances. Aries achieves the same result in 44 seconds (16 times faster) without any initial upper bound. On the `hu/edata` instances, Aries solves two more instances, again without requiring a tight upper bound.

In terms of optimality proofs, our understanding is that the approach would be competitive with CpSat but not with Aries, CPOptimizer or OptalCP and their Failure-Directed Search variants (introduced after this publication). As all experiments in [23] are done with initial upper bounds, we cannot make inferences on the strength of the approach in finding high-quality solutions.

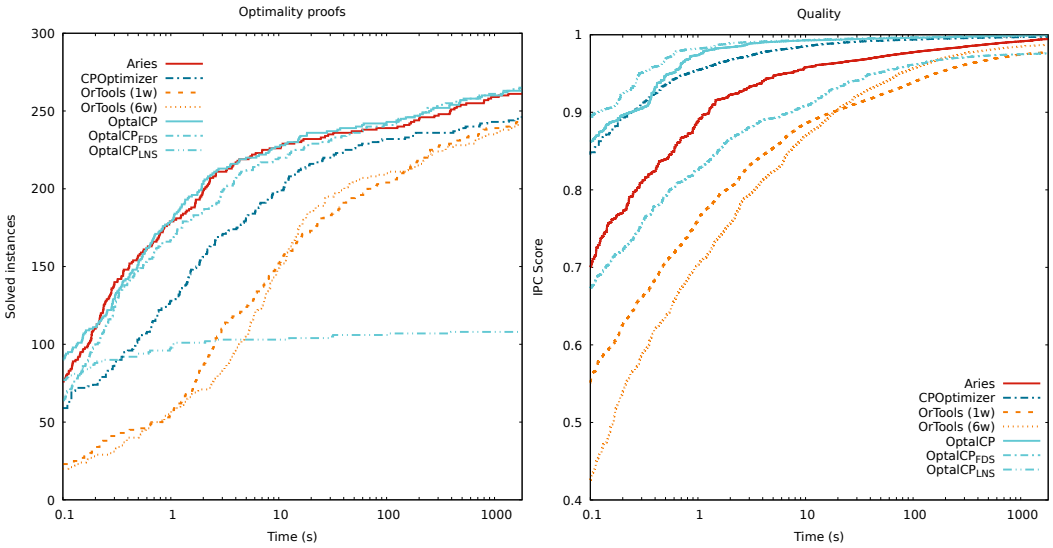

Figure 2 FJS: IPC Score and Solved instances over time for Flexible jobshop (FJS).

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**21**	23	5	**2**	**20**	61	34	**64**	**27**	**4**
CPOPTIMIZER	**21**	15	5	**2**	**20**	61	28	63	**27**	**4**
CPSAT (1w)	20	**24**	5	**2**	19	53	28	61	**27**	**4**
CPSAT (6w)	**21**	18	5	**2**	19	57	26	62	**27**	**4**
OPTALCP	**21**	23	**6**	**2**	**20**	61	**35**	**64**	**27**	**4**
OPTALCP$_{FDS}$	**21**	**24**	**6**	**2**	**20**	**62**	**35**	**64**	**27**	**4**
OPTALCP$_{LNS}$	0	0	4	**2**	9	22	10	31	**27**	3

Figure 3 FJS: Solved instances in 1800s for the difference instance sets of Flexible jobshop (FJS). The best result is highlighted in bold.

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**100.0**	96.9	99.3	99.2	**100.0**	100.0	99.9	**100.0**	100.0	**100.0**
CPOPTIMIZER	**100.0**	99.2	99.7	99.1	**100.0**	99.9	99.8	100.0	100.0	**100.0**
CPSAT (1w)	**100.0**	90.6	98.3	91.5	**100.0**	99.6	99.4	100.0	98.8	**100.0**
CPSAT (6w)	**100.0**	94.9	98.4	96.2	**100.0**	99.9	99.4	99.9	98.9	**100.0**
OPTALCP	**100.0**	99.7	**100.0**	99.9	**100.0**	**100.0**	**100.0**	100.0	100.0	**100.0**
OPTALCP$_{FDS}$	**100.0**	87.0	97.7	95.7	**100.0**	99.9	99.5	100.0	99.6	**100.0**
OPTALCP$_{LNS}$	**100.0**	**100.0**	99.9	**100.0**	**100.0**	100.0	100.0	100.0	**100.0**	**100.0**

Figure 4 FJS: Average IPC Score (×100) after 1800s for the different instance sets of Flexible jobshop (FJS). The best result is highlighted in bold (accounting for the full precision).

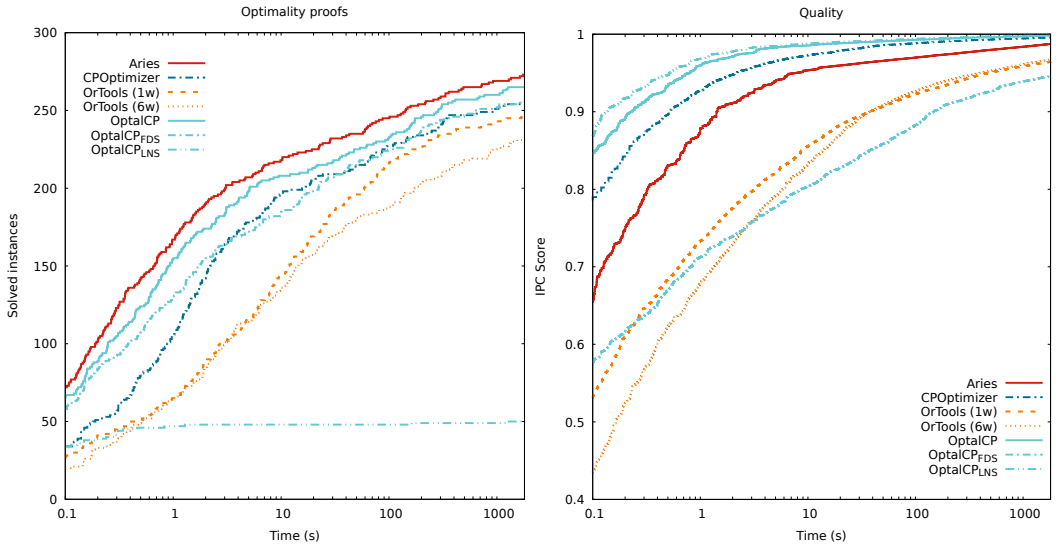

Figure 5 FJS_{TT}: IPC Score and Solved instances over time for Flexible jobshop with transport times (FJS_{TT}).

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**21**	**15**	**10**	1	**20**	**64**	**42**	**65**	**31**	**4**
CPOPTIMIZER	**21**	8	9	**2**	**20**	61	36	**65**	28	**4**
CPSAT (1w)	**21**	**15**	9	0	19	56	37	62	24	**4**
CPSAT (6w)	**21**	7	9	0	19	57	32	64	19	**4**
OPTALCP	**21**	11	9	**2**	**20**	**64**	38	**65**	**31**	**4**
OPTALCP_{FDS}	**21**	12	9	**2**	**20**	62	37	**65**	24	**4**
OPTALCP_{LNS}	0	0	0	0	9	17	0	23	0	1

Figure 6 FJS_{TT}: Solved instances in 1800s for the difference instance sets of Flexible jobshop with transport times (FJS_{TT}). The best result is highlighted in bold.

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**100.0**	92.3	**100.0**	98.8	**100.0**	100.0	**100.0**	**100.0**	99.9	**100.0**
CPOPTIMIZER	**100.0**	98.4	99.7	98.8	**100.0**	99.9	99.7	**100.0**	99.6	**100.0**
CPSAT (1w)	**100.0**	83.5	99.9	88.6	**100.0**	99.7	99.5	100.0	97.4	**100.0**
CPSAT (6w)	**100.0**	84.9	98.9	93.2	100.0	99.9	99.3	**100.0**	96.9	**100.0**
OPTALCP	**100.0**	99.1	99.9	99.8	**100.0**	**100.0**	99.9	**100.0**	99.9	**100.0**
OPTALCP_{FDS}	**100.0**	74.5	99.4	86.1	**100.0**	99.9	99.2	**100.0**	95.3	**100.0**
OPTALCP_{LNS}	100.0	**99.9**	99.8	**99.9**	100.0	100.0	99.9	100.0	**99.9**	**100.0**

Figure 7 FJS_{TT}: Average IPC Score (×100) after 1800s for the different instance sets of Flexible jobshop with transport times (FJS_{TT}). The best result is highlighted in bold (accounting for the full precision).

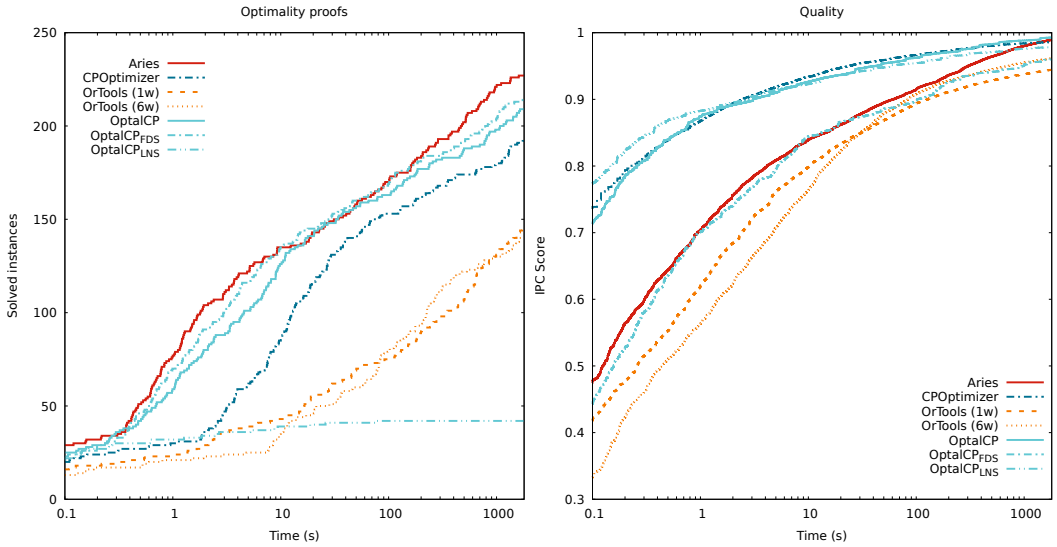

Figure 8 FJS$_{Lag}$: IPC Score and Solved instances over time for Flexible jobshop with maximum time-lag. (FJS$_{Lag}$).

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**20**	23	**5**	**2**	**20**	**47**	27	**50**	29	**4**
CPOPTIMIZER	12	15	**5**	0	**20**	40	30	38	28	**4**
CPSAT (1w)	7	**24**	3	0	19	23	16	27	24	**4**
CPSAT (6w)	7	17	3	0	18	28	9	33	21	**4**
OPTALCP	14	**24**	**5**	0	**20**	40	30	40	32	**4**
OPTALCP$_{FDS}$	14	**24**	**5**	0	**20**	41	**32**	41	**33**	**4**
OPTALCP$_{LNS}$	0	0	2	0	10	1	1	1	24	3

Figure 9 FJS$_{Lag}$: Solved instances in 1800s for the difference instance sets of Flexible jobshop with maximum time-lag. (FJS$_{Lag}$). The best result is highlighted in bold.

	Barnes	BehnkeGeiger	Brandimarte	Dauzere	Fattahi	Hurink/edata	Hurink/rdata	Hurink/sdata	Hurink/vdata	Kacem
ARIES	**100.0**	96.3	**99.7**	95.2	**100.0**	**100.0**	99.4	**99.9**	98.7	**100.0**
CPOPTIMIZER	99.3	98.8	97.9	96.4	**100.0**	98.1	98.9	98.2	99.0	**100.0**
CPSAT (1w)	95.3	88.8	95.0	88.2	100.0	93.6	97.4	95.0	96.1	**100.0**
CPSAT (6w)	97.2	94.8	95.4	87.7	99.9	97.7	96.9	97.9	94.0	**100.0**
OPTALCP	99.6	99.6	99.4	**97.2**	**100.0**	98.8	**99.6**	98.9	**99.7**	**100.0**
OPTALCP$_{FDS}$	99.1	89.3	97.4	79.1	**100.0**	98.5	99.1	98.2	96.0	**100.0**
OPTALCP$_{LNS}$	95.2	**100.0**	98.9	96.7	99.9	96.6	95.7	97.9	99.4	**100.0**

Figure 10 FJS$_{Lag}$: Average IPC Score (×100) after 1800s for the different instance sets of Flexible jobshop with maximum time-lag. (FJS$_{Lag}$). The best result is highlighted in bold (accounting for the full precision).