*Modular Constraint Solver Cooperation via Abstract Interpretation**

PIERRE TALBOT

Interdisciplinary Centre for Security, Reliability and Trust (SNT), University of Luxembourg, Esch-sur-Alzette, Luxembourg (*e-mail:* pierre.talbot@uni.lu)

ÉRIC MONFROY

University of Angers, Angers, France (*e-mail:* eric.monfroy@univ-angers.fr)

CHARLOTTE TRUCHET

University of Nantes, Nantes, France (*e-mail:* charlotte.truchet@univ-nantes.fr)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Cooperation among constraint solvers is difficult because different solving paradigms have different theoretical foundations. Recent works have shown that abstract interpretation can provide a unifying theory for various constraint solvers. In particular, it relies on abstract domains which capture constraint languages as ordered structures. The key insight of this paper is viewing cooperation schemes as abstract domains combinations. We propose a modular framework in which solvers and cooperation schemes can be seamlessly added and combined. This differs from existing approaches such as SMT where the cooperation scheme is usually fixed (*e.g.*, Nelson-Oppen). We contribute to two new cooperation schemes: (i) *interval propagators completion* that allows abstract domains to exchange bound constraints, and (ii) *delayed product* which exchanges over-approximations of constraints between two abstract domains. Moreover, the delayed product is based on delayed goal of logic programming, and it shows that abstract domains can also capture control aspects of constraint solving. Finally, to achieve modularity, we propose the *shared product* to combine abstract domains and cooperation schemes. Our approach has been fully implemented, and we provide various examples on the flexible job shop scheduling problem. Under consideration for acceptance in TPLP.

KEYWORDS: abstract domains, solver cooperation, modularity, constraint programming

1 Introduction

A constraint solver is often more efficient when it targets at a specific constraint language, such as satisfiability (SAT) solvers with Boolean formulas, or linear programming solvers with linear arithmetic constraints. However, problem specifications often consist of constraints of different types. A real-life problem can contain two or more constraints

* This work was partially supported by ANR-15-CE25-0002 Coverif from the French *Agence Nationale de la Recherche*. The *Centre de calcul intensif des Pays de la Loire* (CCIPL) provided the infrastructure to perform the benchmarks. The authors thank the anonymous reviewers for their constructive comments to improve the clarity of the paper. We thank Yinghan Ling for the English proofreading.

such that one is more efficiently treated or it can only be treated in a solver, and the other one in another solver. In such case, it is necessary to find a solver that supports all the constraints of the problem, but it may not be as efficient as specialized solvers. Therefore, the cooperation among solvers becomes a central concern in order to achieve better efficiency and to improve expressiveness of the solvers. Satisfiability modulo theories (SMT) solvers are probably the most well-known cooperation framework as they encapsulate constraint languages in theories that can be combined together by the Nelson-Oppen scheme (Nelson and Oppen, 1979). Lazy clause generation (Ohrimenko et al., 2009) is a more specialized example that mixes SAT solving and propagation-based constraint solvers, and currently it is the state of the art solver for many scheduling problems. On the other end of the spectrum, the black box approaches study the combination of solvers without modifying them (Monfroy, 1998). Overall, the combination of two or more solvers often results in a third solver with little consideration about the modularity and the reuse of its components and its cooperation scheme.

We propose a theoretical and practical framework for constraint solving, where it is possible to introduce new solvers and cooperation schemes in a modular way. In comparison to previous work, our cooperation schemes between solvers are not built in the framework itself, but defined at the same level as solvers. It enables us to define various cooperation schemes among solvers, which can be run concurrently.

Our proposal is based on abstract interpretation (Cousot and Cousot, 1977), a framework to perform static analysis of programs. Abstract domains are an important fragment of abstract interpretation. They capture constraint languages as ordered structures. Abstract interpretation has the advantage to cleanly separate between the logical formula (syntax), the abstract domain (semantics representable in a machine), and the concrete domain (mathematical semantics)—we introduce these concepts in Section 2. This separation and the order theory underlying abstract domains help to prove mathematical properties on the combination of abstract domains. Moreover, abstract domains can be implemented almost directly as they describe the semantics of the solvers.

Contributions This paper focuses on *domain transformers*, which are functors constructing abstract domains from one or more abstract domains. We propose two domain transformers capturing two cooperation schemes. Firstly, the *interval propagators completion* (IPC) which equips any abstract domain with *interval propagators* (Section 3.1). An interval propagator is a function implementing an arithmetic constraint (linear or non-linear). This completion can be applied to products of domains, which results in a cooperation scheme where two domains exchange bound constraints over their shared variables. Secondly, we propose the *delayed product* (DP) which treats a constraint c in an abstract domain A_1 until c becomes treatable in a more efficient abstract domain A_2 (Section 3.2). This technique is inspired by the delayed goal technique of logic programming. The delayed product dynamically rewrites a constraint once its variables are instantiated. In addition, over-approximations of this constraint can be incrementally sent to A_2 before the variables of c are fully instantiated. Finally, we introduce the *shared product* to combine domain transformers sharing abstract domains (Section 3.3). This product enables the hierarchy of abstract domains and transformers to form a directed acyclic graph. We illustrate these abstract domains over the flexible job shop scheduling problem in Section 4. In particular, we reveal that several constraint solvers can be obtained by

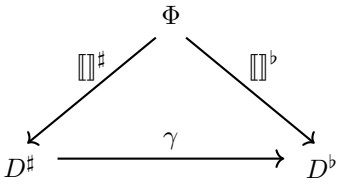
assembling the presented abstract domains, and that they are competitive with state of the art approaches.

Related works Cooperation schemes between the domain of uninterpreted functions (Herbrand universe of a logic program) and various constraint systems have been widely studied in the context of *constraint logic programming* (CLP). For instance, `CLP(BNR)` deals with mixed continuous and discrete domains (Older, 1993). $\mathcal{TOY}$ is a functional CLP language that aims at the solvers cooperation among uninterpreted functions, arithmetic constraints over real numbers, and finite domains (Estévez-Martín et al., 2009). In particular, $\mathcal{TOY}$ introduces the notion of *bridges*, such as `X #==`$_{int,real}$ `Y` between two variables such that X is an integer and Y a real. The transformer IPC can be seen as implementing generic bridges among its underlying domains. A downside of CLP approaches is that the addition of a new constraint system or combination often corresponds to the design of a new language.

The SMT paradigm is an important field about theory combination at the logical level. Theory and abstract domain are two sides of the same coin: theory captures the logical essence of a constraint language, while abstract domain captures its semantics. In fact, it was shown that Nelson-Oppen combination is a specific reduced product, a technique to combine abstract domains, in abstract interpretation (Cousot et al., 2012). Deeper connections have been made by the abstract conflict driven clause learning (ACDCL) framework (D'Silva et al., 2014) which demonstrates that SMT solvers can be considered as fixed point computation over abstract domains. ACDCL is mostly a theoretical proposal and it has not been thoroughly investigated in practice. Overall, cooperation schemes are either built in the theory or left aside in both SMT and ACDCL frameworks.

2 Abstract interpretation for constraint programming

Abstract interpretation is a framework to statically analyze programs by over-approximating the set of values that the variables of the program can take (Cousot and Cousot, 1977). In a nutshell, the following diagram presents the fragment of abstract interpretation we are interested in:



This diagram connects a logical formula, a concrete domain and an abstract domain. The syntax of a program, specifically in our case, of a constraint problem is represented by the set Φ of any quantifier-free first-order logic formulas. We interpret a formula φ to a concrete or abstract domain respectively with $[\![\varphi]\!]^\flat$ and $[\![\varphi]\!]^\sharp$. The concrete domain represents the mathematical semantics of this formula, its exact set of solutions which may be infinite and not computer-representable. The abstract domain corresponds to the machine semantics of this formula that might under- or over-approximate the set of solutions of the concrete domain. Approximations are particularly insightful on continuous domains, such as real numbers, which have to be approximated using floating point

numbers. An abstract domain is connected to the concrete domain by a concretization function $\gamma : D^\sharp \to D^\flat$, which is useful to prove properties of the abstract domain[1]. In the following, as we mainly manipulate abstract domains, we will omit the $\sharp$ symbol on the operators, for instance $[\![.]\!]^\sharp$ is written as $[\![.]\!]$. This section summarizes previous work (Pelleau et al., 2013; Talbot et al., 2019) in which more formal definitions and proofs can be found.

Concrete domain A constraint satisfaction problem (CSP) is a tuple (X, D, C) where X is a set of variables, $D = D_1 \times \ldots \times D_n$ the sets of values taken by each variable $x_i \in X$, and C a set of relations over variables, called *constraints*. A constraint $c \in C$, defined on the variables $x_1, \ldots, x_n$ is satisfied when $c(v_1, \ldots, v_n)$ holds for all $v_i \in D_i$. The *concrete domain* is the powerset lattice $D^\flat = \langle \mathcal{P}(D), \supseteq \rangle$ ordered by inclusion. The concrete interpretation function maps a CSP (X, D, C)[2] to an element in $D^\flat$ representing its set of solutions:

$$[\![(X, D, C)]\!]^\flat = \{(D'_1, \ldots, D'_n) \mid D'_i \subseteq D_i \text{ and all } c \in C \text{ satisfied}\}$$

Abstract domain In abstract interpretation, an abstract domain is a partially ordered set equipped with useful operations for programs analysis. This notion has been adapted to constraint programming, where some operators are reused (*e.g.*, join and interpretation function) and some are new (*e.g.*, state and split) for its application to constraint solving. In the following, "abstract domain" will refer to this modified notion of abstract domain for constraint programming. The set $K = \{\mathit{true}, \mathit{false}, \mathit{unknown}\}$ represents elements of Kleene logic, in which we have $\mathit{false} \wedge \mathit{unknown} = \mathit{false}$ and $\mathit{true} \wedge \mathit{unknown} = \mathit{unknown}$.

Definition 1 (*Abstract domain*)
An abstract domain for constraint programming is a lattice $\langle A, \leq \rangle$ where A is a set of computer-representable elements equipped with the following operations:

- $\bot$ is the smallest element and, if it exists, $\top$ the largest.
- $\sqcup : A \times A \to A$ is called the *join*, it performs the union of the information contained in two elements.
- $\gamma : A \to D^\flat$ is a monotonic *concretization* function mapping an abstract element to its set of solutions.
- $\mathit{state} : A \to K$ gives the state of an element: *true* if the element satisfies all the constraints of the abstract domain, *false* if at least one constraint is not satisfied, and *unknown* if satisfiability cannot be established yet.
- $[\![.]\!] : \Phi \to A$ is a partial function transferring a formula to an element of the abstract domain[3]. This function is not necessarily defined for all formulas since an abstract domain efficiently handles a delimited constraint language.

[1] Abstract interpretation usually relies on an abstraction function $\alpha : D^\flat \to D^\sharp$. In our case, the concrete solutions set is fixed and always given by $[\![\varphi]\!]^\flat$, thus we have $\alpha([\![\varphi]\!]^\flat) = [\![\varphi]\!]^\sharp$.
[2] Note that (X, D, C) is just a structured presentation of a logical formula.
[3] Alternatively, this function could be total and every unsupported formula mapped to $\bot$ which is a correct over-approximation. However, it prevents us from distinguishing between tautological formulas (since $[\![\mathit{true}]\!] = \bot$) and unsupported formulas. In the first case, we wish to interpret the formula in A, while in the second case we prefer to look for another, more suitable, abstract domain.

- *closure* : $A \to A$ is an extensive function ($\forall x, x \leq \mathit{closure}(x)$) which eliminates inconsistent values from the abstract domain.
- *split* : $A \to \mathcal{P}(A)$ divides an element of an abstract domain into a finite set of sub-elements.

We refer to the ordering of the lattice L as $\leq_L$ and similarly for any operation defined on L, unless no confusion is possible. An abstract element $a \in A$ under-approximates the concrete solutions set of a formula φ if $\gamma(a) \subseteq [\![\varphi]\!]^\flat$, which implies that all points in a are solutions, but solutions might be missing. Dually, a over-approximates φ if $\gamma(a) \supseteq [\![\varphi]\!]^\flat$, which implies that all solutions are preserved but there might be non-solution points in a. We can prove properties on abstract domains by verifying these two equations.

We present an algorithm to refine the approximation of an element $[\![\varphi]\!] \in A$ approximating a formula φ. This algorithm is generic over an abstract domain A.

```
1: function solve(a ∈ A)
2:     a ← closure(a)
3:     if state(a) = true then return {a}
4:     else if state(a) = false then return {}
5:     else
6:         ⟨a_1, ..., a_n⟩ ← split(a)
7:         return ⋃_{i=0}^{n} solve(a_i)
8:     end if
9: end function
```

This algorithm follows the usual solving pattern in constraint programming which is *propagate and search*. We infer as much information as possible with *closure*, and then divide the problem into sub-problems with *split*. We rely on *state* for the base cases defined when we reach a solution or an inconsistent node. We obtain the solutions of a constraint set C in an abstract domain A with $\mathit{solve}(\bigsqcup_{c \in C} [\![c]\!])$. It is noteworthy that the abstract domain $a \in A$ can be a composition of several abstract domains through domain transformers (see Section 3). The over-approximation property extends to `solve` with $\bigcup\{\gamma(a) \mid a \in \texttt{solve}([\![\varphi]\!])\} \supseteq [\![\varphi]\!]^\flat$, and dually for under-approximation. A termination condition and proof of this algorithm are given in (Talbot et al., 2019). We illustrate these definitions in some abstract domains as follows.

Box abstract domain We denote by $\mathbb{A}$ the set of integers $\mathbb{Z} \cup \{-\infty, \infty\}$. An interval is a pair $(l, u) \in \mathbb{A}^2$ of the lower and upper bounds, written as $[l..u]$, defined as $\gamma([l..u]) = \{x \in \mathbb{Z} \mid l \leq x \leq u\}$. The set of intervals $I = \langle \{[l..u] \mid \forall l, u \in \mathbb{A}\}, \leq, \bot, \top, \sqcup \rangle$ is a lattice ordered by set inclusion $\leq \triangleq \supseteq$. It has a bottom element $\bot \triangleq [-\infty, \infty]$, a top element $\top \triangleq \{\}$, and a join $\sqcup \triangleq \cap$ defined by set intersection. An interval can be used to represent the domain of a single variable. In order to represent collection of variable's domains, we consider the lattice of partial functions $[V \nrightarrow I]$ from the set of variable's names V to the lattice of intervals I. Practically, elements of $[V \nrightarrow I]$ can be thought as arrays of interval domains. This lattice is studied by Fernández and Hill (2004) for constraint solving in a more general setting. The *box abstract domain* $\mathcal{B} = \langle [V \nrightarrow I], \leq \rangle$ equips $[V \nrightarrow I]$ with the operators of Def. 1. Boxes capture a small constraint language consisting of the constraints $x \leq b$, $x \geq b$, $x < b$, $x > b$ and $x = b$, where $x \in V$, $b \in \mathbb{A}$. The role of the interpretation function is then to map

each supported constraint to an element of the abstract domain. The logical conjunction coincides with the join in the lattice. For instance we have $B = [\![x > 2 \wedge x \leq 4 \wedge y > 0]\!] = [\![x > 2]\!] \sqcup [\![x \leq 4]\!] \sqcup [\![y > 0]\!] = \{x \mapsto [3..\infty]\} \sqcup \{x \mapsto [-\infty..4]\} \sqcup \{y \mapsto [1..\infty]\} = \{x \mapsto [3..4], y \mapsto [1..\infty]\}$. The concrete set of elements is obtained by listing all solutions, *e.g.*, $\gamma(B) = \{\{(x,3),(y,1)\},\{(x,4),(y,1)\},\{(x,3),(y,2)\},\ldots\}$. We observe that this set is infinite, which is why we need an abstract domain approximating this set in a finite way. In the case of boxes, the interpretation function is both an under- and over-approximation because, for all formulas φ such that $[\![\varphi]\!]$ is defined, we have $\gamma([\![\varphi]\!]) \subseteq [\![\varphi]\!]^\flat$ and $\gamma([\![\varphi]\!]) \supseteq [\![\varphi]\!]^\flat$. Therefore, once a constraint has been interpreted, we have the best possible approximation, and thus *closure* is simply the identity function. It is not always the case, as explained below with octagons. An element $b \in \mathcal{B}$ is consistent if $\gamma(a) \neq \{\}$, which boils down to the observation that no variable has an empty interval. This observation can be used to define *state*, which is always equal to either *true* or *false*. The *split* operator can only be useful when boxes are used in combination with other abstract domains. It can be defined by selecting a variable $x = [l..u]$ in $b \in \mathcal{B}$ and dividing this interval into two parts, *e.g.*, $split(b) = \{b \sqcup [\![x = l]\!], b \sqcup [\![x > l]\!]\}$. It is worth mentioning that many different *split* operators are possible, which are more or less efficient depending on the problem at hand.

Octagon abstract domain The octagon abstract domain (Miné, 2006), denoted by O, is more expressive than boxes because it can interpret constraints of the form $\pm x \pm y \leq c$ and $\pm x \leq c$ where x, y are variables and c is a constant (either over integers, floating point numbers or rational numbers). Internally, an octagon is represented by a difference-bound matrix of size $\mathcal{O}(n^2)$ where n is the number of variables. Its closure operator is the Floyd-Warshall algorithm which runs in $\mathcal{O}(n^3)$ in the general case. An incremental version in $\mathcal{O}(n^2)$ is available when only one constraint is added.

A domain transformer: logic completion The logic completion $\mathsf{L}(A)$ is a domain transformer: it takes an abstract domain A as a parameter and produces a new abstract domain supporting logical connectors over the constraint language of A. For example, the formula $c_1 \triangleq (x = 1 \vee x = 2)$ is neither interpretable in boxes nor in octagons, but it is in $\mathsf{L}(\mathcal{B})$ or $\mathsf{L}(\mathsf{O})$. In the presence of disjunction, we have $state(closure([\![c_1]\!])) = unknown$, because there is too few information to infer whether $x = 1$ or $x = 2$. A choice must be made, and this is where the *split* operator and the `solve` algorithm become necessary. The problem is decomposed into two subproblems $x = 1$ and $x = 2$ which are solved in turn. The union of their solutions is the solutions set of the initial problem.

Combination of domains: direct product The value of this abstract framework stands out when abstract domains are combined. For instance, consider the formula $c_2 \triangleq (x > 4 \wedge x < 7) \Rightarrow y + z \leq 4$. The abstract domain $\mathsf{L}(\mathsf{O})$ is expressive enough to interpret c_2. However, the constraints on x can be treated more efficiently in boxes than in octagons due to the lower space complexity of boxes. Therefore, it is advantageous to interpret $x > 4 \wedge x < 7$ in a box and $y + z \leq 4$ in an octagon. In order to achieve that, we rely on the direct product $\mathcal{B} \times \mathsf{O}$. When stacked with the logic completion transformer, it gives us $\mathsf{L}(\mathcal{B} \times \mathsf{O})$. We define the direct product as follows.

Definition 2 (*Direct product*)
Let $A_1, \ldots, A_n$ be a collection of n abstract domains. The direct product is an abstract domain $\langle A_1 \times \ldots \times A_n, \leq \rangle$ where each operator is defined coordinatewise, *e.g.*, $(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n) \Leftrightarrow \bigwedge_{1 \leq i \leq n} a_i \leq_i b_i$, with $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in A_1 \times \ldots \times A_n$.

There is a small issue about the previous formula c_2: the constraint $x > 4 \wedge x < 7$ is interpretable both in boxes and octagons. In this case, the direct product will interpret this formula in both domains, which is not the behavior we expect. To solve this problem, we annotate formulas with an integer, denoted as $\varphi : \mathtt{i}$, meaning that φ should be interpreted in the i^{th} component of the product. Formally, we have $[\![\varphi\colon \mathtt{i}]\!] \triangleq (\bot_1, \ldots, [\![\varphi]\!]_i, \ldots, \bot_n)$. The formula can be duplicated as many times as needed to be interpreted in more than one domain. If we annotate c_2 with $(x > 4 \wedge x < 7)\colon \mathtt{1} \Rightarrow (y + z \leq 4)\colon \mathtt{2}$, the first constraint will be interpreted in boxes and the second one in octagons—note that the logic completion forwards the interpretation of annotated sub-formulas to the underlying domain, here the product.

The cooperation happening between the box and octagon domains in $\mathsf{L}(\mathcal{B} \times \mathsf{O})$ is fully logical. This form of cooperation allows us to address some complex problems, as shown in (Talbot et al., 2019). However, as soon as two constraints belonging to different abstract domains share variables, the variable's domains (*e.g.*, intervals) are not shared among the domains. Indeed, the operators of the direct product defined coordinatewise, each $closure_i$ is independently applied to each component of the product, but the new information obtained is never exchanged. In the next section, we propose two domain transformers that exchange information between domains in two different ways. As a cross-product, we show that domain transformers also capture operational aspects (such as delayed goals), that are more difficult to express in a fully logical setting.

3 Domain transformers for cooperation schemes

3.1 Interval propagators completion

Consider the constraint $c_3 \triangleq x > 1 \wedge x + y + z \leq 5 \wedge y - z \leq 3$. The constraint $x > 1$ can be interpreted in boxes and $y - z \leq 3$ in octagons, but $x + y + z \leq 5$ is too general to be interpreted in any abstract domain we introduced until now. Moreover, the last constraint shares a variable with the other two. The interval propagators completion is a domain transformer, denoted as $\mathsf{IPC}(A)$, which solves both problems at once. $\mathsf{IPC}(A)$ extends the constraint language of any abstract domain A to arbitrary arithmetic constraints. The constraint $(x > 1)\colon \mathtt{1} \wedge x + y + z \leq 5 \wedge (y - z \leq 3)\colon \mathtt{2}$ can be fully interpreted in $\mathsf{IPC}(\mathcal{B} \times \mathsf{O})$. To understand how IPC proceeds, we must first introduce two new concepts: the projection function and propagators.

$\mathsf{IPC}(A)$ expects A to provide an additional projection function of the variables onto intervals, defined as $project : (A \times V) \to I$. The function $project(a, x)$ must over-approximate the set of solutions of x in a, *i.e.*, for each value v that takes x in $\gamma_A(a)$, $v \in \gamma_I(project(a, x))$. The interval lattice might be defined over rational numbers $\mathbb{Q}$, floating point numbers $\mathbb{F}$ or integers $\mathbb{Z}$ depending on A. Projection can be implemented directly in many arithmetic domains such as boxes and octagons, but it is sometimes more difficult as it is the case in polyhedra. In the case of the direct product, projection

is defined as $project((a_1, \ldots, a_n), x) = project_1(a_1, x) \sqcup \ldots \sqcup project_n(a_n, x)$. If the variable x does not belong to an abstract domain, $project_i(a_i, x)$ maps to $\bot$. The projection is defined on integer intervals if any of the underlying abstract domains projects x onto integers—since integers are more constrained than other types. In the cases of $\mathbb{F}$ and $\mathbb{Q}$, rational numbers are preferred as they are more precise.

A propagator on an abstract domain A is an extensive function $p : A \to A$ implementing an inference algorithm for a given constraint. The *closure* operator of any abstract domain A can be viewed as a propagator on A. The difference is that a propagator implements a single constraint whereas abstract domains support a larger constraint language. To illustrate propagators, we consider a propagator for the constraint $x \geq y$ generically on an abstract domain A with a projection function.

$$[\![x \geq y]\!] = p_{\geq} = \lambda a.a \sqcup_A [\![x \geq y_\ell]\!]_A \sqcup_A [\![y \leq x_u]\!]_A$$

with $project(a, x) = [x_\ell..x_u]$ and $project(a, y) = [y_\ell..y_u]$. For instance, given $project(a, x) = [1..2]$ and $project(a, y) = [2..3]$, and the constraint $x \geq y$, we obtain $project(p_{\geq}(a), x) = [2..2]$ and $project(p_{\geq}(a), y) = [2..2]$. We notice that this propagation step is extensive since we have $a \leq p_{\geq}(a)$. The constraint $x + y + z \leq 5$ can be implemented by a similar propagator. The propagation performed on x, y and z will be automatically communicated to the direct product $\mathcal{B} \times \mathsf{O}$, which in turn will communicate the new bounds to the box and octagon components. However, we must solve a small technical issue. In the constraint c_3, we do not wish to propagate new bounds on x in octagons since there is no octagonal constraint involving x. In a propagator defined similarly to $p_{\geq}$, the variable x would be added in the octagon. We overcome this issue with a function $embed : A \times A \to A$ defined as $embed(a_1, a_2) = a_1 \sqcup a_2$ if $vars(a_2) \subseteq vars(a_1)$[4], and $embed(a_1, a_2) = a_1$ otherwise. We define this function coordinatewise on the direct product. The corrected version of the propagator $p_{\geq}$ is given as follows:

$$[\![x \geq y]\!] = p_{\geq} = \lambda a.embed_A(a, [\![x \geq y_\ell]\!]_A) \sqcup_A embed_A(a, [\![y \leq x_u]\!]_A)$$

Besides extensiveness, we usually require a propagator to over-approximate the set of solutions (soundness), *i.e.*, it should not remove solutions of the logical constraint, in order to guarantee the correctness of the solving algorithm, formally $\gamma(p(a)) \supseteq [\![\varphi]\!]^\flat$. Finally, we associate to each propagator p a $state_p$ function which is defined similarly to the one of abstract domain. In particular, an element a is a solution of p if $state_p(a) = true$.

Putting all the pieces together, we obtain the lattice $Pr = \langle \mathcal{P}(Prop), \subseteq \rangle$ where $Prop$ is the set of all propagators (extensive and sound functions). The *interval propagators completion* of an abstract domain A with projection is given by the Cartesian product $\mathsf{IPC}(A) = \langle A \times Pr, \leq \rangle$ with its operations defined as follows for $(a, P), (a', P') \in \mathsf{IPC}(A)$:

- $(a, P) \leq (a', P') \Leftrightarrow a \leq_A a' \wedge P \subseteq P' \qquad (a, P) \sqcup (a', P') \triangleq (a \sqcup_A a', P \cup P')$.
- $state((a, P)) \triangleq state_A(a) \wedge \bigwedge_{p \in P} state_p(a)$ which means that we reach a solution when a is a solution for all propagators in P.
- $\gamma((a, P)) \triangleq \bigcup \{\gamma_A(a') \mid a' \geq_A a \wedge state((a', P)) = true\}$.

[4] The function $vars : A \to \mathcal{P}(V)$ can be generically added to any abstract domain by capturing the variables of a formula φ before it is interpreted into an element of A.

- The function $[\![c]\!]$ associates the constraint c to its propagator p_c and state function $state_p$. For example, we can rely on the propagation algorithm `HC4` (Benhamou et al., 1999) which works generically over arbitrary arithmetic constraints.
- $closure((a, \{p_1, \ldots, p_n\})) \triangleq (\mathbf{fp}(p_1 \circ \ldots \circ p_n)(a), \{p_1, \ldots, p_n\})$.
- $split((a, P)) \triangleq \{(a', P) \mid a' \in split_A(a)\}$.

The *propagation step* is realized by computing a fixed point (**fp**) of $p_1 \ldots p_n$ altogether in *closure*. We do not require to compute the *least* fixed point as it has no impact on the termination property of the solving algorithm. There are many possible implementations of *closure* as shown in (Apt, 1999). The next lemma explains that IPC over a direct product of abstract domains results in a sound over-approximation.

Lemma 3
Let A_1 and A_2 be abstract domains, and φ a logic formula. If A_1 and A_2 over-approximate the set of solutions $[\![\varphi]\!]^\flat$, then $\mathsf{IPC}(A_1 \times A_2)$ also over-approximates $[\![\varphi]\!]^\flat$.

Proof
Let $\bar{i} = 3 - i$. We know that $project_i$ maps to an over-approximated interval of its variables. This interval view is transferred into $A_{\bar{i}}$ via $[\![v \geq l_i \wedge v \leq u_i]\!]_{\bar{i}}$ which over-approximates the constraints as well. Therefore, only over-approximations are involved during the information exchange and no solution is lost. □

3.2 Delayed product

IPC is only able to exchange bound constraints although there are often opportunities for stronger cooperation between domains. We present the *delayed product*, a product inspired by delayed goals in logic programming, that dynamically exchanges specialized constraints between two domains. We consider again the constraint $x + y + z \leq 5$ in the formula c_3. Whenever the variable x becomes instantiated, meaning that $x = v$ for a value v, we can rewrite the constraint to $y + z \leq 5 - v$ and interpret it in octagons for additional propagation.

Let A_1 and A_2 be abstract domains such that A_1 is strictly more expressive[5] than A_2, but A_2 is supposed to be more efficient on its constraints language. The delayed product $\mathsf{DP}(A_1, A_2)$ evaluates a set of formulas $F \subseteq \Phi$ into A_1 until they become instantiated enough to be supported in A_2. A variable x is instantiated in $a \in A$ whenever $fix(a, x) \triangleq (x_\ell = x_u)$, with $project(a, x) = [x_\ell..x_u]$, holds. For readability, we write $val(a, x) = v$ with v the value of x in a whenever $fix(a, x)$ holds. To describe this product, we rely on a rewriting function that replaces every instantiated variable with its value, formally defined as:

$$\varphi \rightarrow_a \begin{cases} \varphi[x \rightarrow val(a, x)] & \text{if } \exists x \in vars(\varphi),\ fix(a, x) \\ \varphi & \text{otherwise} \end{cases}$$

A formula to be transferred is an element of the lattice $\mathsf{FT} = [\Phi \nrightarrow Bool]$ where $Bool = \{true, false\}$ and $false \leq true$. Let $f \in \mathsf{FT}$, then $f(\varphi)$ is *true* if the formula has already been transferred, and *false* otherwise. We write $nt(f) = \{\varphi \mid f(\varphi) = false\}$ the set of

[5] The constraint language supported by the interpretation function of A_2 is included in A_1.

non-transferred formulas. The delayed product $\mathsf{DP}(A_1, A_2) = \langle A_1 \times A_2 \times \mathsf{FT}, \leq \rangle$ is an abstract domain inheriting most operations from the Cartesian product. The different operations are defined as follows:

- $[\![\varphi]\!] \triangleq \begin{cases} (\bot_1, [\![\varphi]\!]_2, \{\}) & \text{if } [\![\varphi]\!]_2 \text{ is defined} \\ ([\![\varphi]\!]_1, \bot_2, \{\varphi \mapsto \mathit{false}\}) & \text{otherwise} \end{cases}$
- $\mathit{closure}((a_1, a_2, c)) \triangleq (a_1, a_2, c) \sqcup \bigsqcup_{\varphi \in nt(C)} \mathit{closure\_one}(a_1, a_2, \varphi)$

  with $\mathit{closure\_one}(a_1, a_2, \varphi) \triangleq \begin{cases} (a_1, a_2 \sqcup_2 [\![\varphi']\!]_2, \{\varphi \mapsto \mathit{true}\}) \text{ where } \varphi \rightarrow^*_{a_1} \varphi' \\ \qquad \text{if } [\![\varphi']\!]_2 \text{ is defined and } \mathit{vars}(\varphi') \subseteq \mathit{vars}(a_2) \\ (a_1, a_2, \{\varphi \mapsto \mathit{false}\}) \end{cases}$

The condition $\mathit{vars}(\varphi') \subseteq \mathit{vars}(a_2)$ in *closure_one* restricts the product to add a constraint only if the variables of the constraint are already defined in the domain. It enables the user of the domain to decide with better flexibility which variables need to be instantiated before the constraint is transferred.

Improved closure By over-approximating a constraint c, it is possible to interpret it in A_2 even before it becomes instantiated enough. For instance, the constraint $x+y+z \leq 5$ can be over-approximated to $y + z \leq 5 - x_u$ with $\mathit{project}(a_1, x) = [x_\ell..x_u]$ since the maximal value that x can ever take is its upper bound. Let x be a variable in $a_1 \in A_1$, e an arithmetic expression, and $\mathit{project}(a_1, x) = [x_\ell..x_u]$. We rely on the following rewriting function $\twoheadrightarrow$:

$$x \leq e \twoheadrightarrow_{a_1} x_\ell \leq e \qquad\qquad x \geq e \twoheadrightarrow_{a_1} x_u \geq e$$

Lemma 4
The function $\twoheadrightarrow$ over-approximates the constraints $x \leq e$ and $x \geq e$.

Proof
For any value v of x, if $v \leq e$ is entailed, then $l \leq e$ is also entailed since $l \leq v \leq e$ (similarly for $x \geq e$). □

We extend the definition of closure to take into account these over-approximations:

$$\mathit{closure\_one}(a_1, a_2, \varphi) \triangleq \begin{cases} (a_1, a_2 \sqcup_2 [\![\varphi']\!]_2, \{\varphi \mapsto \mathit{true}\}) \text{ where } \varphi \rightsquigarrow_{a_1} \varphi' \\ \qquad \text{if } [\![\varphi']\!]_2 \text{ is defined and } \mathit{vars}(\varphi') \subseteq \mathit{vars}(a_2) \\ (a_1, a_2 \sqcup_2 [\![\varphi']\!]_2, \{\varphi \mapsto \mathit{false}\}) \text{ where } \varphi \twoheadrightarrow_{a_1} \varphi' \\ \qquad \text{if } [\![\varphi']\!]_2 \text{ is defined and } \mathit{vars}(\varphi') \subseteq \mathit{vars}(a_2) \\ (a_1, a_2, \{\varphi \mapsto \mathit{false}\}) \end{cases}$$

In the case of a partial transfer, the formula φ is not set to *true* since it is not yet fully taken into account into A_2.

3.3 Combining domain transformers

In order to complete our cooperation framework, we tackle the case where two domain transformers share abstract domains. For instance, consider the formula $c_4 \triangleq (x = 0 \vee x = 1) \wedge x * y \leq 5$. We can interpret $x = 0 \vee x = 1$ in $\mathsf{L}(\mathcal{B})$, which supports bound constraints with disjunctions, and $x * y \leq 5$ in $\mathsf{IPC}(\mathcal{B})$. If we combine these two domains

in a direct product $\mathsf{L}(\mathcal{B}) \times \mathsf{IPC}(\mathcal{B})$, the underlying box domain of each transformer will not be shared, because the direct product does not exchange information among its components. Conversely, sometimes it is important for efficiency to keep two abstract elements of the same type separated. Consider the example of two octagons with each n distinct variables, the closure operator has a complexity of $\mathcal{O}(n^3) + \mathcal{O}(n^3)$ when two octagon elements are created, but $\mathcal{O}((n+n)^3)$ when merged. Therefore, both possibilities of either merging or keeping the domains separated must be available. To this aim, we propose the *shared product* which is a direct product with named components and sharing among components. To make the notation explicit, we define an element of the shared product as a list of abstract domain declarations. As an example, the previous domain with (**D1**) and without (**D2**) a shared box are written as:

$$\begin{array}{ll} \mathbf{D1} = & \mathcal{B}\ box; \\ & \mathsf{L}(\mathcal{B})\ lbox(box); \\ & \mathsf{IPC}(\mathcal{B})\ ipc(box); \end{array} \qquad \begin{array}{ll} \mathbf{D2} = & \mathsf{L}(\mathcal{B})\ lbox(\bot_{\mathcal{B}}); \\ & \mathsf{IPC}(\mathcal{B})\ ipc(\bot_{\mathcal{B}}); \end{array}$$

The line $\mathsf{L}(\mathcal{B})\ lbox(box)$; indicates that the underlying box domain of $\mathsf{L}(\mathcal{B})$ is shared and given by the element *box*. We say that *box* is a dependency of *lbox*. Every element must be declared before being used as dependencies. When no dependency is expected, the parameter is an unnamed bottom element, *e.g.*, $\bot_{\mathcal{B}}$. In that case, the boxes underlying $\mathsf{L}(\mathcal{B})$ and $\mathsf{IPC}(\mathcal{B})$ are not shared. In order to define the shared product, we rely on two functions to respectively project and join the dependencies:

$$\pi : A \to A_1 \times \ldots \times A_n \qquad\qquad \kappa : A \times A_1 \times \ldots \times A_n \to A$$

In the delayed product, we have $\pi((a_1, a_2, c)) = (a_1, a_2)$ and $\kappa((a_1, a_2, c), d_1, d_2) = (a_1 \sqcup d_1, a_2 \sqcup d_2, c)$. We now define the shared product.

Definition 5 (*Shared product*)
The shared product $\langle A_1\ x_1(d_1^1, \ldots, d_m^1)\ ;\ \ldots\ ;\ A_n\ x_n(d_1^n, \ldots, d_m^n), \leq \rangle$ is a direct product $A_1 \times \ldots \times A_n$ in which the *closure* operator is interleaved with a reduction operator. Let a_i be an element of the product and $\pi_i(a_i) = (b_j, \ldots, b_k)$ the dependencies of a_i, where $b_\ell = \bot$ if $d_\ell^i = \bot$, for all $j \leq \ell \leq k$. Then each ρ_i is an idempotent and monotone function defined as:

$$\rho_i(a_1, \ldots, a_n) = (a_1, \ldots, a_j \sqcup b_j, \ldots, a_k \sqcup b_k, \ldots, \kappa_i(a_i, a_j, \ldots, a_k), \ldots, a_n)$$

We define ρ as the fixed point of $\rho_1 \circ \ldots \circ \rho_n$. This reduction operator is applied when computing the closure: $closure((a_1, \ldots, a_n)) \triangleq \rho(closure_1(a_1), \ldots, closure_n(a_n))$. The interpretation function can be extended to support named constraints:

$$[\![c{:}x]\!] = (\bot_1, \ldots, [\![c]\!]_i, \ldots, \bot_n) \text{ where } x = x_i$$

which is simpler to read than the index notation of the direct product.

We illustrate the two roles of ρ_i with an example. Let the element a be $(box, lbox, ipc) \in$ **D1**. Consider $\rho_2(box, lbox, ipc) = (box \sqcup \pi(lbox), \kappa(lbox, box), ipc)$, which merges *lbox* with the rest of the product. First, ρ_2 merges the dependency of *lbox* into *box* with $box \sqcup \pi(lbox)$. Second, ρ_2 updates the dependency of *lbox* with *box* using $\kappa(lbox, box)$. In general, since we compute a fixed point of ρ, which is also the least by the Knaster-Tarski fixed point theorem, the abstract domains and domain transformers are totally merged.

In practice, the dependencies are implemented by using pointers. Therefore, π and κ are defined implicitly for all abstract domains. As in the former example, at any time a new information is available in *box*, it is automatically accessible to both $\mathsf{L}(\mathcal{B})$ and $\mathsf{IPC}(\mathcal{B})$ due to the sharing via pointers.

An advantage of this framework is that no effort is required by a domain transformer to be plugged into the shared product. Moreover, the transformers are fully compositional w.r.t. the shared product, *i.e.*, they can be combined with any other transformers without being modified. We will illustrate the shared product in a larger example in the next section.

4 Case study and evaluation

Flexible job shop scheduling Job shop scheduling is a well-known NP-hard combinatorial problem. We have n jobs and m machines such that a job $1 \leq j \leq n$ is a series of T_j tasks that must be scheduled on distinct machines in turn. For each job j and task $1 \leq t \leq T_j$, the duration of the task is written as $d_{j,t} \in \mathbb{Z}$, and the machine on which the task t is performed is written as $m_{j,t} \in \{1, \ldots, m\}$. The variables of the problem are the starting dates $s_{j,t}$ for every task t. For each job, we must ensure that every task is finished before the next one starts (precedence constraints):

$$\forall 1 \leq j \leq n,\ \forall 1 \leq t \leq T_j - 1,\ s_{j,t} + d_{j,t} \leq s_{j,t+1} \tag{1}$$

Two tasks of two different jobs must not use the same machine at the same time:

$$\begin{aligned} \forall 1 \leq i < j \leq n,\ \forall 1 \leq t \leq T_i,\ \forall 1 \leq u \leq T_j \\ m_{i,t} = m_{j,u} \Rightarrow s_{i,t} + d_{i,t} \leq s_{j,u} \vee s_{j,u} + d_{j,u} \leq s_{i,t} \end{aligned} \tag{2}$$

The disjunctive constraints ensure each pair of tasks (t, u) using the same machine do not overlap. Usually, the goal is to find a schedule of the tasks finishing as early as possible. Therefore, it is an optimization problem that seeks to minimize the makespan.

$$\forall 1 \leq j \leq n,\ s_{j,T_j} + d_{j,T_j} \leq \mathit{makespan} \tag{3}$$

The flexible job shop scheduling problem (Brucker and Schlie, 1990) generalizes the job shop scheduling to multiple machines. A task can be scheduled on a possible set of machines which might have different processing times for the same task. The model is now parametrized by a set of possible machines $M_{j,t} \subseteq \{1, \ldots, m\}$ for each task t, and by a duration $dur_{j,t,m} \in \mathbb{Z}$ depending on the task *and* the machine. The parameter $m_{j,t}$ of the job shop problem becomes a decision variable $m_{j,t} \in M_{j,t}$ modeling on which machine every task t is run. The duration of a task depends on the machine on which it is run, thus every $d_{j,t}$ becomes a decision variable as well:

$$\forall 1 \leq j \leq n,\ \forall 1 \leq t \leq T_j,\quad \bigvee_{k \in M_{j,t}} m_{j,t} = k \wedge d_{j,t} = dur_{j,t,k} \tag{4}$$

Constraints (1), (2) and (3) stay syntactically the same but over decision variables instead of parameters.

Crafting abstract domains for the flexible job shop The abstract domain $\mathsf{L}(\mathsf{IPC}(\mathcal{B}))$ is expressive enough to treat the full flexible job shop scheduling problem. However, as

expected it is not very efficient. Octagons are more efficient than boxes on precedence constraints. To achieve that, we build an abstract domain, that we name $\mathbf{FJS_1}$, based on boxes and octagons:

$$\begin{array}{l}\mathcal{B}\ box;\\ \mathsf{O}\ oct;\\ \mathsf{L}(\mathsf{IPC}(\mathcal{B}\times\mathsf{O}))\ any((box, oct));\end{array}$$

We note the usage of nested parenthesis $((box, oct))$ in order to define the dependencies of nested abstract domains. To ease the distribution of constraints in abstract domains, we declare *box* and *oct* although they are not shared. In the case of *oct*, there is another subtlety: it is necessary to declare *oct* otherwise its *closure* operator will not be called since IPC does not call the closure of its underlying domain. The next step is to distribute each constraint in the components of $\mathbf{FJS_1}$. At the first sight, octagons are of limited interest because all precedence constraints are defined on three variables. Nevertheless, for most instances of the flexible job shop, we observe that some tasks can only be executed on one machine, or some tasks take the same time on all machines. Hence, some precedence constraints are immediately octagonal since the duration is fixed. Constraints in Eq. (1) are distributed in $\mathbf{FJS_1}$ as follows:

$$\forall 1 \leq j \leq n,\ \forall 1 \leq t \leq T_j - 1, \begin{cases} (s_{j,t} + d' \leq s_{j,t+1}){:}oct & \text{if } \{d'\} = \{dur_{j,t,k} \mid k \in M_{j,t}\} \\ (s_{j,t} + d_{j,t} \leq s_{j,t+1}){:}any & \text{otherwise} \end{cases}$$

It is the same for Eq. (3). All the others constraints can be interpreted in *any*. In addition, since IPC relies on the underlying domain to represent the variable's domains, we must add all the variables in the box domain first, for each job j and task t:

$$(s_{j,t} \leq h \wedge makespan \leq h \wedge m_{j,t} \leq max(M_{j,t}) \wedge d_{j,t} \leq max(\{dur_{j,t,k} \mid k \in M_{j,t}\})){:}box$$

The constant h represents the horizon, which is the latest date at which a task can start.

In $\mathbf{FJS_1}$, we statically dispatch the precedence constraints when creating the model. Because of the delayed product, we can dynamically dispatch the precedence constraints when the durations become fixed, that is, during the solving process. Precedence constraints can be solved efficiently in the domain $\mathbf{PREC} = \mathsf{DP}(\mathsf{IPC}(\mathcal{B}\times\mathsf{O}), \mathsf{O})$. The precedence constraints with three variables are interpreted in $\mathsf{IPC}(\mathcal{B}\times\mathsf{O})$, similarly to $\mathbf{FJS_1}$. In addition, exact and over-approximations of precedence constraints with two variables are dynamically sent in the octagon element thanks to the delayed product. To experiment with this idea, we craft the abstract domain $\mathbf{FJS_2}$ as follows:

$$\begin{array}{ll}\mathcal{B}\ box; & \\ \mathsf{O}\ oct; & \\ \mathbf{PREC}\ prec(((box, oct)), oct); & \text{precedence constraints Eq. (1) and (3)}\\ \mathsf{L}(\mathcal{B}\times\mathbf{PREC})\ no\_overlap(box, prec); & \text{non-overlap constraints Eq. (2)}\\ \mathsf{L}(\mathcal{B})\ alternatives(box); & \text{machine alternatives constraints Eq. (4)}\end{array}$$

The constraints can be annotated with the name of the relevant abstract domains, similarly to what we did for $\mathbf{FJS_1}$. The formula in Eq. (2) is constituted of an implication and disjunctions that can be interpreted in the abstract domain *no_overlap*. The atoms of the formula are either equality constraints ($m_{i,t} = m_{j,u}$) that can be interpreted by the box element $\mathcal{B}$, or precedence constraints that can be interpreted in **PREC**. Finally,

Eq. (4) could be interpreted in *no_overlap*, but since **PREC** is not useful for this formula, we can avoid the unnecessary indirection by interpreting this formula in the dedicated *alternatives* domain.

Implementation and evaluation We have implemented the abstract domains and transformers presented above in the constraint solver AbSolute (Pelleau et al., 2013), which is programmed in OCaml and available online[6]. Our experiments are replicable and all the results are also publicly available. One of our design goals was to keep the solver as close as possible to its underlying theory. To achieve this goal, we relied on OCaml functors, such that each domain transformers is a functor parametrized by its sub-domains. See Appendix A for an example of the OCaml code modeling the abstract domain $\mathbf{FJS_1}$.

The experiments are all performed on an Intel(R) Xeon(TM) E5-2630 V4 running at 2.20GHz on GNU Linux. We evaluate three solvers: AbSolute v0.10, GeCode v6.1 (Schulte et al., 2019) which is a state of the art propagation-based constraint solver, and Chuffed v0.10.4 (Ohrimenko et al., 2009) which is a hybrid solver between constraint propagation and SAT solving. Chuffed shows excellent results on scheduling problems including the flexible job shop (Schutt et al., 2013). Since we primarily focus on evaluating the propagation process, we selected a search strategy available in all solvers. This strategy, that we call `dms`, assigns the domain of each variable to its lower bound and selects the variables with the smallest domain first (*first-fail strategy*). Furthermore, we first assign all durations, then all machines, and finally the starting dates variables. We experimented on two sets of instances, named `edata` and `rdata`, due to Hurink et al. (1994), which are still challenging today (Schutt et al., 2013). The difference among the sets is the average ratio of machines available per task, `edata` has few machines per task, and for `rdata` most tasks can be scheduled on several machines. Each solver is run once on each instance for a maximum of 10 minutes.

The results are exposed in Table 1. For each solver, we read in column Δ_{LB} the percentage of how far is the obtained solution from the best known lower bound. For example, the value **66** in bold in Table 1 indicates that Chuffed found 66 strictly better bounds than GeCode.

Firstly, although AbSolute is only a prototype, we observe that on `edata` it finds 36 bounds that are better than the ones found by GeCode, and 23 bounds better than Chuffed. This demonstrates that communication between domains brings a computational advantage. For data sets with more machines, the efficiency of AbSolute drops behind the other solvers. This is because we do not treat machines in a special way in contrast to GeCode or Chuffed that use a *cumulative* global constraint.

Secondly, the difference between FJS_1 and FJS_2 is less obvious since FJS_2 is only able to find a few better bounds. This is explained by `dms` which fixes the durations at the top of the search tree, thus all over-approximations are exchanged early in the search, and do not impact the propagation of most of the nodes. However, for the flexible job shop, `dms` was the best strategy we tried in AbSolute. Nevertheless, we found that FJS_2 was able to find its best bound 20% quicker than FJS_1 w.r.t. the number of nodes for about 90% of the instances. This confirms that better cooperation leads to better pruning in general.

[6] The version of AbSolute used in this paper is accessible at `github.com/ptal/AbSolute/tree/iclp2020`.

solver	$\Delta_{LB}(\%)$	**FJS$_1$**	**FJS$_2$**	GeCode	Chuffed	$\Delta_{LB}(\%)$	**FJS$_1$**	**FJS$_2$**	GeCode	Chuffed
			`edata`					`rdata`		
FJS$_1$	20.4	□	0	36	23	46.4	□	0	4	0
FJS$_2$	20.4	1	□	36	23	46.4	2	□	4	0
GeCode	20.9	30	30	□	0	31.7	61	61	□	0
Chuffed	12.2	43	43	**66**	□	24.2	66	66	66	□

Table 1: Experiments on the flexible job shop scheduling problem (2 * 66 instances).

We believe that this framework achieves modularity because new abstract domains can be seamlessly combined with existing ones in order to treat new constraints. Besides, the presented abstract domains and transformers *are not specifically designed* for the jobshop scheduling problem. These transformers are applicable to numerous other problems. Ziat et al. (2019) combine boxes and polyhedra to solve continuous constraint problems; their product is a particular instance of our delayed product. Furthermore, the delayed product could also be applied to car sequencing problems which involve linear constraints and octagonal constraints (Brand et al., 2007).

5 Conclusion and future work

Abstract constraint solving is an exciting new area of research where the foundation of constraint solving is reformulated as abstract interpretation. We contribute to this area by developing a modular abstract framework allowing solvers and cooperation schemes to be combined seamlessly. To this end, we have introduced the *interval propagators completion* and the *delayed product* domain transformers implementing two cooperation schemes. Moreover, we have introduced the *shared product* to modularly combine domain transformers.

There are three important perspectives of this work. The first one is to catch up with ACDCL by incorporating conflict learning in AbSolute, which is crucial for efficiency as notably demonstrated by lazy clause generation in Chuffed (Ohrimenko et al., 2009). Secondly, an inference mechanism to automatically build the right abstract domain to solve a logical formula would be interesting. This is not trivial as a formula might be interpretable in several abstract domains, thus expressiveness and efficiency must be taken into account in the inference process. Finally, it is most often necessary to program a customized search strategy in order to achieve better solving efficiency. This framework only supports combination of search strategies in a restricted way. We suggest to rely on *spacetime programming*, a synchronous and concurrent search strategy language operating over lattice structures to integrate search in this framework (Talbot, 2019).

References

Apt, K. R. 1999. The essence of constraint propagation. *Theoretical computer science 221,* 1-2, 179–210. `https://doi.org/10.1016/S0304-3975(99)00032-8`.

Benhamou, F., Goualard, F., Granvilliers, L., and Puget, J.-F. 1999. Revising hull and box consistency. In *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming.* MIT press, 230–244. `https://doi.org/10.7551/mitpress/4304.003.0024`.

Brand, S., Narodytska, N., Quimper, C.-G., Stuckey, P., and Walsh, T. 2007. Encodings of the sequence constraint. In *International conference on principles and practice of constraint programming*. Springer, 210–224. https://doi.org/10.1007/978-3-540-74970-7_17.

Brucker, P. and Schlie, R. 1990. Job-shop scheduling with multi-purpose machines. *Computing 45,* 4 (Dec.), 369–375. https://doi.org/10.1007/BF02238804.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 238–252. https://doi.org/10.1145/512950.512973.

Cousot, P., Cousot, R., and Mauborgne, L. 2012. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM (JACM) 59,* 6, 31. https://doi.org/10.1145/2395116.2395120.

D'Silva, V., Haller, L., and Kroening, D. 2014. Abstract satisfaction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, San Diego, California, USA, 139–150. https://doi.org/10.1145/2535838.2535868.

Estévez-Martín, S., Hortalá-González, T., Rodríguez-Artalejo, M., del Vado-Vírseda, R., Sáenz-Pérez, F., and Fernández, A. J. 2009. On the cooperation of the constraint domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{F}$ in CFLP. *Theory and Practice of Logic Programming 9,* 4, 415–527. https://doi.org/10.1017/S1471068409003780.

Fernández, A. J. and Hill, P. M. 2004. An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems 26,* 1 (Jan.), 1–46. https://doi.org/10.1145/963778.963779.

Hurink, J., Jurisch, B., and Thole, M. 1994. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum 15,* 4, 205–215. https://doi.org/10.1007/BF01719451.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC) 19,* 1, 31–100. https://doi.org/10.1007/s10990-006-8609-1.

Monfroy, E. 1998. An environment for designing/executing constraint solver collaborations. *Electronic Notes in Theoretical Computer Science 16,* 1, 1 – 22. https://doi.org/10.1016/S1571-0661(05)80588-2.

Nelson, G. and Oppen, D. C. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS) 1,* 2, 245–257. https://doi.org/10.1145/357073.357079.

Ohrimenko, O., Stuckey, P. J., and Codish, M. 2009. Propagation via lazy clause generation. *Constraints 14,* 3 (Sept.), 357–391. http://dx.doi.org/10.1007/s10601-008-9064-x.

Older, W. 1993. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*. 239–249.

Pelleau, M., Miné, A., Truchet, C., and Benhamou, F. 2013. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 434–454. https://doi.org/10.1007/978-3-642-35873-9_26.

Schulte, C., Tack, G., and Lagerkvist, M. 2019. *Modeling and Programming with Gecode*.

Schutt, A., Feydy, T., and Stuckey, P. J. 2013. Scheduling optional tasks with explanation. In *Principles and Practice of Constraint Programming*. 628–644. https://doi.org/10.1007/978-3-642-40627-0_47.

Talbot, P. 2019. Spacetime Programming: A Synchronous Language for Composable Search Strategies. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP 2019)*. ACM, New York, NY, USA, 18:1–18:16. https://doi.org/10.1145/3354166.3354183.

Talbot, P., Cachera, D., Monfroy, E., and Truchet, C. 2019. Combining Constraint Languages via Abstract Interpretation. In *31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2019)*. Portland, USA, 50–58. https://doi.org/10.1109/ICTAI.2019.00016.

Ziat, G., Maréchal, A., Marie, P., Miné, A., and Truchet, C. 2019. Combination of Boxes and Polyhedra Abstractions for Constraint Solving. In *The 8th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2019)*. Porto, Portugal.

Appendix A $\mathbf{FJS_1}$ in AbSolute

We give an example of how to turn $\mathbf{FJS_1}$ into an abstract domain at the implementation-level. We first create the leaves of the combination, in this case the box and octagon abstract domains:

```
module Box = Box_base(Box_split.First_fail_LB)(Bound_int)
module Octagon = Octagon.Make(ClosureHoistZ)(Octagon_split.MSLF)
```

These two domains are parametrized by a *split* operator, we further indicate that we need a box over integers, and an octagon over integers as well—`ClosureHoistZ` is a possible implementation of the closure operator for octagon. We now encapsulate these domains in the interval propagator completion:

```
module BoxOct = Direct_product(Prod_cons(Box)(Prod_atom(Octagon)))
module IPC = Propagator_completion(Box.Vardom)(BoxOct)
```

The completion is additionally parametrized by a variable domain (here the same as box) which indicates the domain in which the propagation takes place. For instance, if we have a completion over integers and floating point numbers (*i.e.*, $\mathsf{IPC}(\mathcal{B}(\mathbb{Z}) \times \mathcal{B}(\mathbb{F}))$), the constraints could be evaluated in a rational domain since it subsumes both integers and floating point numbers. The completion takes care of the required conversions.

The remaining step is to derive the logic completion of IPC, and to gather all components in the shared product:

```
module LC = Logic_completion(IPC)
module FJS = Shared_product(
  Prod_cons(BoxOct)(
  Prod_cons(IPC)(
  Prod_atom(LC))))
```

This product can then be instantiated with empty abstract domains, and solved with a fixed point algorithm as presented in Section 1.

This demonstrates that abstract domains are composed in a modular way at the theoretical level, but also at the implementation level.