A Propagator Design Framework for Constraints over Sequences

Jean-Noël Monette and Pierre Flener and Justin Pearson
Uppsala University, Dept of Information Technology
751 05 Uppsala, Sweden
firstname.lastname@it.uu.se

Abstract

Constraints over variable sequences are ubiquitous and many of their propagators have been inspired by dynamic programming (DP). We propose a conceptual framework for designing such propagators: pruning rules, in a functional notation, are refined upon the application of transformation operators to a DP-style formulation of a constraint; a representation of the (tuple) variable domains is picked; and a control of the pruning rules is picked.

1 Introduction

Many combinatorial problems have constraints over variable sequences. A lot of propagators inspired by dynamic programming (DP) techniques have been proposed for either specific such constraints (e.g., Knapsack (Trick 2003), Spread and Deviation (Pesant 2011)) or, more often, families of constraints that can be expressed in a generic way (e.g., Automaton (Beldiceanu, Carlsson, and Petit 2004), Regular (Pesant 2004), Cost-regular (Demassey, Pesant, and Rousseau 2006), Slide (Bessiere et al. 2008), Seqbin (Petit, Beldiceanu, and Lorca 2011; Katriel, Narodytska, and Walsh 2012), Cost-MDD (Gange, Stuckey, and Van Hentenryck 2013), and Regularcount (Beldiceanu et al. 2014)). Although their propagators look very different from each other, many of them are derived from a few common abstract recipes.

In this paper, we show that such propagator design recipes can be made explicit and encoded in a compact manner. Our main contribution is a conceptual framework for designing propagators on variable sequences (Sections 2 and 6). It offers operators for the stepwise refinement of pruning rules starting from a formulation of the constraint (Section 4), choices for the representation of the variable domains (Section 5), and choices for the control of the set of pruning rules (Section 6). We illustrate our framework using published propagators (Section 3) and alternative new ones (Section 7). In Section 8, we conclude and outline future work. The supplemental material referred to in the body of the text can be found at http://www.it.uu.se/research/group/astra/publications/AAAI14-DP-Appendix.pdf.

2 A Propagator Design Framework

We consider global constraints that can be formulated in DP style as a conjunction of the following constraints:

\[ P_F(A_0^1 \ldots A_n^0, F^1 \ldots F^f) \]  (C_F)
\[ P_L(A_1^1 \ldots A_n^1, B_1^1 \ldots B_n^1, A_{i-1}^0 \ldots A_{i-1}^0) \quad i \in 1..n \]  (C_I)
\[ P_L(A_1^a \ldots A_n^a, L_1^1 \ldots L_r^f) \]  (C_L)

where

- \( A_i^1 \ldots A_i^n \) are variables, for \( i \) in \( 0..n \), called link variables as they appear in two constraints.
- \( B_i^1 \ldots B_i^n \) are variables, for \( i \) in \( 1..n \), called local variables as they appear in only one constraint \( C_i \).
- \( F^1 \ldots F^f \) are variables appearing only in constraint \( C_F \).
- \( L^1 \ldots L_r^f \) are variables appearing only in constraint \( C_L \).
- \( P_F, P_L \), and the \( P_i \) are predicate symbols.

We call this a DP formulation. Many constraints have DP formulations, including all those mentioned in Section 1.

Example 1. Deviation \((X_1 \ldots X_n, m, D)\) holds iff the average of variables \(X_1 \ldots X_n\) is the integer \(m\) and the sum of their deviations from \(m\) is variable \(D\) (i.e., \( \sum_{i=1}^n |X_i - m| = D \)). A DP formulation is:

\[ S_0 = 0 \land D_0 = 0 \]  (C_F)
\[ S_i = S_{i-1} + X_i \land \]  \( i \in 1..n \)  (C_I)
\[ D_i = D_{i-1} + |X_i - m| \]
\[ S_n = m \cdot n \land D_n = D \]  (C_L)

where variables \( S_i \) and \( D_i \) are introduced to represent partial sums and partial deviations. The link variables are the \( S_i \) and \( D_i \) (so \( a = 2 \)), the local variables are the \( X_i \) (so \( b = 1 \)), there is no variable appearing only in \( C_F \) (so \( f = 0 \)), and \( D \) is the only variable appearing only in \( C_L \) (so \( l = 1 \)).

After presenting a generic propagator for DP formulations, we discuss its axes of parametrisation.

Generic Propagator

A direct implementation of a DP formulation can miss pruning if \( a > 1 \) because the underlying constraint network is then not Berge-acyclic (Beeri et al. 1983). The bundling of the link variables would make the network Berge-acyclic.
Depending on the representation of the domains of the resulting tuple variables and the implementation of the individual constraints, domain consistency can then be achieved.

Our generic propagator thus assumes the usage of tuple variables. A tuple variable is a variable whose domain is a set of tuples (see Section 5 for further details). After introducing a tuple variable \( A_i \) for each \( i \) in \( 0..n \) to represent the tuple \( A_1 \times A_2 \times \cdots \times A_n \) of link variables, we reformulate the DP formulation using the tuple variables instead of the variables they represent. Additional constraints, denoted by \( C_{A_i} \), link each tuple variable with its link variables.

**Example 2.** For **Deviation** (see Example 1), introducing a tuple variable \( SD_i \) for \( S_i \) and \( D_i \) gives the new formulation:

\[
SD_0 = \langle 0, 0 \rangle \quad (C_F)
\]

\[
SD_i = SD_{i-1} + \langle X_i, |X_i - m| \rangle \quad i \in 1..n \quad (C_i)
\]

\[
SD_n = \langle m \cdot n, D \rangle \quad (C_L)
\]

\[
SD_i = \langle S_i, D_i \rangle \quad i \in 0..n \quad (C_{A_i})
\]

where \(+\) and \(-\) are set union in a component-wise manner, and \( \langle v, w \rangle \) builds a tuple composed of \( v \) and \( w \).

The generic propagator is a set of generic pruning rules: for each constraint \( c \) in a DP formulation and for each variable \( v \) appearing in \( c \), prune the domain of \( v \) based on \( c \) and the current domains of the other variables appearing in \( c \).

**Design of Propagators**

The generic propagator can be specialised along three largely orthogonal axes:

- The set of pruning rules.
- The representation of the domains of the tuple variables.
- The control of the set of pruning rules.

We propose a framework to describe and design propagators through variation along those three axes.

To express a pruning rule in a high-level fashion, we introduce the function \( \text{smap_o_filter}(f, \phi, T) \), which filters a tuple set \( T \) according to a condition \( \phi \) and then maps it to another set using a function \( f \), which takes a tuple and returns a tuple set: \( \text{smap_o_filter}(f, \phi, T) = \bigcup_{t \in T \land \phi(t)} f(t) \). (The ‘o’ in the function name stands for the ‘o’ of function composition of the set mapping \( f \) and the tuple filter \( \phi \).) This function can express many pruning rules. In Section 4 we introduce transformation operators for the stepwise refinement of pruning rules starting from a DP formulation of a constraint.

In Section 5, we bring together some existing representations of tuple variables, and propose a notation to combine them, thereby making explicit their design space.

The design space for the control of the set of pruning rules is very large. In Section 6, we focus on three very common approaches.

### 3 The Design of Published Propagators

All published propagators mentioned in Section 1 can be designed within our framework. Before discussing the framework in detail in Sections 4 to 6, we illustrate it by two examples. For ease of exposition, all notation has been simplified for the needs of these examples: the full syntax and semantics of the tuple operators are given in the online supplemental material, as are the full examples wherever they have been abridged or simplified.

**Example 3.** For **Deviation**(\( X_1 \cdots X_n, m, D \)), introduced in Example 1, a propagator aiming at domain consistency in \( O(n^2 d u) \) time is given by (Pesant 2011), where \( d \) is the size of the largest domain of the \( X_i \), denoted \( \text{dom}(X_i) \), and \( u \) is the size of the union of these domains.

The domain of a link tuple variable \( SD_i \) (from the DP formulation introduced in Example 2) is represented as a mapping from values of the partial sum \( S_i \) to intervals of values for the partial deviation \( D_i \). We write this mapping \( E \rightarrow I \), where \( E \) denotes an extensional representation (all values of \( S_i \) are used), and \( I \) denotes an interval representation (for each value of \( S_i \) only the bounds of \( D_i \) are maintained): see Section 5 for details. The propagator is incremental and stateful through the use of a layered graph.

The pruning of \( SD_i \) based on \( C_i \) is given by (Pesant 2011) using a recurrence relation (we omit the base cases):

\[
\ell(i, s_i) = \min_{x_i \in \text{dom}(X_i)} (\ell(i - 1, s_i - x_i) + |x_i - m|)
\]

\[
u(i, s_i) = \max_{x_i \in \text{dom}(X_i)} (u(i - 1, s_i - x_i) + |x_i - m|)
\]

for \( i \in 1..n \) and \( s_i \in \text{dom}(S_i) \), where \( \ell(i, s_i) \) and \( u(i, s_i) \) are lower and upper bounds on the value of \( D_i \) when \( S_i \) takes value \( s_i \). We can express this pruning in our framework as a function of the current domains of the variables involved in \( C_i \) and returning the new domain of \( SD_i \):

\[
\text{smap_o_filter}(
\lambda \langle x_i, s_{i-1} \rangle . \{ s_{i-1} + x_i \} \times (\rho_{s_{i-1}}(SD_{i-1}) + \{|x_i - m|\}),
\lambda t . \text{true},
\text{dom}(X_i) \times \pi_1(SD_{i-1}))
\]

where \( \lambda v . b \) is an anonymous function with argument \( v \) and body \( b \), while \( \pi_1(Y) \) is the projection of the domain of tuple variable \( Y \) onto its \( j \)-th component (i.e., \( \{ \langle t_j | \langle t_1 \cdots t_j \rangle \in \text{dom}(Y) \} \}) and \( \rho_1(Y) \) is the selection of second components paired up with \( t \) as first component in the domain of the 2-tuple variable \( Y \) (i.e., \( \{ t_2 | \langle t, t_2 \rangle \in \text{dom}(Y) \} \}). Hence, the function above can be described as taking all pairs composed of a value \( x_i \) in \( \text{dom}(X_i) \) and a value \( s_{i-1} \) appearing as first component in \( \text{dom}(SD_{i-1}) \), and returning all the pairs composed of \( s_{i-1} + x_i \) and \( d_{i-1} + |x_i - m| \) for each value \( d_{i-1} \) paired with \( s_{i-1} \) in \( \text{dom}(SD_{i-1}) \).

We now show how to obtain this pruning rule in stepwise fashion in our framework. All transformation operators are explained in Section 4. **Instantiation** from the definition of constraint \( C_i \) in the DP formulation of Example 2 gives:

\[
\text{smap_o_filter}(
\lambda \langle x_i, s_{i-1}, d_{i-1} \rangle . \{ s_{i-1} \},
\lambda \langle x_i, s_{i-1}, d_{i-1} \rangle . s_{i-1} = d_{i-1} + \langle x_i, |x_i - m| \rangle,
\text{dom}(X_i) \times \text{dom}(SD_i) \times \text{dom}(SD_{i-1}))
\]

This rule is inefficient, as it iterates over all tuples in the Cartesian product of the domains. The **functionalisation** of \( SD_i \) replaces all occurrences of \( s_{i-1} \) by its functional definition and removes \( \text{dom}(SD_i) \) from the Cartesian product:
\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_i, s_{i-1} \rangle \cdot \{ s_{i-1} + x_i \} \times \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle d_{i-1} \rangle \cdot \{ d_{i-1} + |x_i - m| \}, \lambda t. \text{true}, \rho_{s_{i-1}}(\text{SD}_{i-1})) \), \lambda t. \text{true}, \text{dom}(X_i) \times \text{dom}(\text{SD}_{i-1}) ) \]

The embedding of the second component of SD\(_{i-1}\) (representing D\(_{i-1}\)) introduces an inner \texttt{smap}_\circ \texttt{filter} expression that iterates over \(\rho_{s_{i-1}}(\text{SD}_{i-1})\) for each value \(s_{i-1}\) in \(\pi_1(\text{SD}_{i-1})\); also, it projects the outer Cartesian product:

\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_i, s_{i-1} \rangle \cdot \{ s_{i-1} + x_i \} \times \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle d_{i-1} \rangle \cdot \{ d_{i-1} + |x_i - m| \}, \lambda t. \text{true}, \rho_{s_{i-1}}(\text{SD}_{i-1})) \), \lambda t. \text{true}, \text{dom}(X_i) \times \pi_1(\text{SD}_{i-1}) ) \]

The setification of the second component of SD\(_{i-1}\) replaces all occurrences of \(d_{i-1}\) by the set \(\rho_{s_{i-1}}(\text{SD}_{i-1})\), assuming a pointwise lifting of value operations to set operations; also, it removes that set from the inner Cartesian product:

\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_i, s_{i-1} \rangle \cdot \{ s_{i-1} + x_i \} \times \lambda t. \text{true}, \rho_{s_{i-1}}(\text{SD}_{i-1})) \), \lambda t. \text{true}, \text{dom}(X_i) \times \pi_1(\text{SD}_{i-1}) ) \]

Simplification of the inner \texttt{smap}_\circ \texttt{filter} yields the rule of (Pesant 2011) above. Similarly, pruning rules are refined for the other variables of C\(_i\) and the other constraints of the DP formulation. None of these transformations are specific to DEVIATION, and they can be used for other constraints. \(\square\)

**Example 4.** \textsc{SeqBin}(\(X_0 \ldots X_n, S, B, D\)) holds iff \(S\) is the number of times the binary constraint \(B\) holds for pairs of successive variables in the sequence \(X_0 \ldots X_n\) (i.e., \(S = \sum_{i=1}^n [B(X_{i-1}, X_i)]\), where \([\gamma]\) is 1 if constraint \(\gamma\) holds and 0 otherwise) and the binary constraint \(D\) holds for all pairs of successive variables (i.e., \(\land_{i=1}^n D(X_{i-1}, X_i)\)). Through appropriate instantiation of \(B\) and \(D\), constraints such as \textsc{IncreasingValues} and \textsc{Change} (Beldiceanu et al. 2007) can be formulated using \textsc{SeqBin}. After introduction of partial sum variables \(S_i\), a DP formulation is:

\[
S_0 = 0 \quad (C_P)
\]
\[
S_i = S_{i-1} + [B(X_{i-1}, X_i)] \land D(X_{i-1}, X_i) \] \quad \text{\(i \in [1..n]\)} \quad (C_1)
\]
\[
S_n = S \quad (C_L)
\]

The link variables are the \(X_i\) and \(S_i\) (so \(a = 2\)) and there is no local variable (so \(b = 0\)). For using the framework, we introduce a tuple variable XS\(_i\) for each pair of \(X_i\) and \(S_i\).

This constraint was introduced by (Petit, Beldiceanu, and Lorca 2011) with a propagator based on (\(E \rightarrow I\)) representation of the link tuple variables. (Katsirelos, Narodytska, and Walsh 2012) argue this representation is insufficient to achieve domain consistency, and prove that by replacing the intervals (\(I\)) by another representation, namely \(i\)-zipped sets (denoted by \(I\)), one achieves domain consistency. We denote this second representation by \(E \rightarrow I\). Except for the domain representations, both papers use the same non-incremental stateless propagator. For conciseness, we only show how to refine one pruning rule, namely for the link tuple variable XS\(_i\) based on the definition of constraint C\(_i\) in the DP formulation: the refinement is similar or even simpler for the other variables and constraints. Instantiation from the definition of C\(_i\) gives:

\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_{i-1}, s_{i-1} \rangle \cdot \{ x_{i-1}, s_{i-1} \} \cdot \{ x_i, s_i \}, \lambda t. \text{true}, \rho_{s_{i-1}}(\text{XS}_{i-1})) \), \lambda t. \text{true}, \text{dom}(X_{i-1}) \times \pi_1(\text{XS}_{i-1}) ) \]

The functiona\(l\)isation of the second component of XS\(_{i-1}\) (representing S\(_{i-1}\)) replaces all occurrences of \(s_{i-1}\) by its definition, which functionally depends on other values, and projects onto the first component of the Cartesian product:

\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_{i-1}, s_{i-1} \rangle \cdot \{ x_{i-1}, s_{i-1} \} \cdot \{ x_i, s_i \}, \lambda t. \text{true}, \rho_{s_{i-1}}(\text{XS}_{i-1})) \), \lambda t. \text{true}, \text{dom}(X_{i-1}) \times \pi_1(\text{XS}_{i-1}) ) \]

To avoid iterating over all domain values of the second component of XS\(_{i-1}\) (representing S\(_{i-1}\)), its embedding and setification give, after simplification:

\[ \text{\texttt{smap}_\circ \texttt{filter}}( \lambda \langle x_{i-1}, x_i \rangle \cdot \{ x_{i-1} \} \cdot \rho_{s_{i-1}}(\text{XS}_{i-1})) \), \lambda t. \text{true}, \rho_{s_{i-1}}(\text{XS}_{i-1})) \), \lambda t. \text{true}, \pi_1(\text{XS}_{i-1}) \times \pi_1(\text{XS}_{i-1}) ) \]

which is exactly the rule described by (Petit, Beldiceanu, and Lorca 2011; Katsirelos, Narodytska, and Walsh 2012).

When \(d\) is the size of the largest domain of the \(X_i\), the published propagators take \(O(nd^2)\) time, and, if \(B\) is row convex, \(O(nd)\) time by computing some intermediate data structures. Future work includes incorporating such a property in the design of a propagator. \(\square\)

### 4 The Refinement of Pruning Rules

We introduced in Section 2 the \texttt{smap}_\circ \texttt{filter} function to express in a high-level way the pruning rules of the generic propagator. As exemplified in Section 3, it is possible to refine pruning rules from the DP formulation of a constraint using a small set of transformation operators, defined next.

The instantiation operator generates, from the definition of a constraint \(P(Y_1, \ldots, Y_p)\) in a DP formulation, a pruning rule \texttt{smap}_\circ \texttt{filter}(\(f, \phi, T\)) for pruning a variable \(Y_j\) based on \(P\), with \(j \in [1..p]\); the tuple set \(T\) is the Cartesian product of the domains of \(Y_1, \ldots, Y_p\); the filter \(\phi\) tests if a tuple satisfies \(P\); and \(f\) maps a tuple to a singleton containing its \(j^{th}\) component. This is written:
The pruning rule for $Y_j$ generated by the instantiation operator is in general inefficient: the aim of the remaining transformation operators is to reduce the time complexity of a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$ by reducing the arity of its Cartesian product $T$. These operators target some variable $Y_k$, with $k$ not necessarily equal to $j$.

The functionalisation operator exploits a functional dependency of (a component of) a (tuple) variable $Y_k$ on other variables in a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$ for $P(Y_1,\ldots,Y_p)$, with $k \in 1..p$, by dropping the iteration over $\text{dom}(Y_k)$. The $k$th component of the argument tuple of mapping $f$ and filter $\phi$ is replaced by its functional definition, and Cartesian product $T$ is projected accordingly. For example, the functionalisation of $V$ at position $k = 1$ transforms

$$\text{smap}_o\text{filter}(\lambda\langle y_1,\ldots,y_p\rangle \cdot \{y_j\},\
\lambda\langle y_1,\ldots,y_p\rangle \cdot P(y_1,\ldots,y_p),\
\text{dom}(Y_1) \times \cdots \times \text{dom}(Y_p))$$

Examples of instantiation are given in Section 3.

The pruning rule for $Y_j$ generated by the instantiation operator is in general inefficient: the aim of the remaining transformation operators is to reduce the time complexity of a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$ by reducing the arity of its Cartesian product $T$. These operators target some variable $Y_k$, with $k$ not necessarily equal to $j$.

The functionalisation operator exploits a functional dependency of (a component of) a (tuple) variable $Y_k$ on other variables in a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$ for $P(Y_1,\ldots,Y_p)$, with $k \in 1..p$, by dropping the iteration over $\text{dom}(Y_k)$. The $k$th component of the argument tuple of mapping $f$ and filter $\phi$ is replaced by its functional definition, and Cartesian product $T$ is projected accordingly. For example, the functionalisation of $V$ at position $k = 1$ transforms

$$\text{smap}_o\text{filter}(\lambda\langle v, w\rangle \cdot \{v\},\
\lambda\langle v, w\rangle \cdot v = 5 \cdot w, \text{dom}(V) \times \text{dom}(W))$$

into

$$\text{smap}_o\text{filter}(\lambda\langle w\rangle \cdot \{5 \cdot w\}, \lambda t \cdot \text{true}, \text{dom}(W))$$

using the functional dependency of $V$ on $W$.

The setification operator removes the domain of (a component of) a (tuple) variable $Y_k$ from the Cartesian product $T$ of a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$, and replaces the $k$th component of the argument tuple of mapping $f$ and filter $\phi$ by $\text{dom}(Y_k)$. Operations on values are lifted pointwise to operations on sets: this enables the use of efficient implementations of operations for specific representations of tuple variables (see Section 5). For example, the setification of $W$ at position $k = 2$ transforms

$$\text{smap}_o\text{filter}(\lambda\langle v, w\rangle \cdot \{v + w\},\
\lambda\langle v, w\rangle \cdot v \leq w, \text{dom}(V) \times \text{dom}(W))$$

into

$$\text{smap}_o\text{filter}(\lambda\langle v\rangle \cdot \{v + \text{dom}(W)\},\
\lambda\langle v\rangle \cdot v \leq \text{dom}(W), \text{dom}(V))$$

The meaning of $v \leq \text{dom}(W)$ is that $v$ is at most some element of $\text{dom}(W)$ (i.e., $\exists w \in W \cdot v \leq w$): this can be simplified (using operators mentioned below) into $v \leq \max(\text{dom}(W))$.

The embedding operator splits (a component of) a (tuple) variable off the Cartesian product $T$ of a pruning rule $\text{smap}_o\text{filter}(f,\phi,T)$, by moving it to a new $\text{smap}_o\text{filter}$ expression inside the definition of the mapping $f$. The rest of $T$ is kept to generate the tuples. This operator is used to avoid projecting a tuple variable onto its Cartesian product when applying other operators, typically setification. For instance, if a tuple variable $V$ has two components, then its domain can be split into two parts: the first component stays in $T$, which becomes $\pi_1(V)$; the second component is moved to an inner $\text{smap}_o\text{filter}$ expression with Cartesian product $\rho_1(V)$, where $v$ is the first component of the argument tuple of mapping $f$ and filter $\phi$. Examples are given in Section 3.

Other transformation operators can be necessary to apply the previous ones. For instance, the order of operands in a commutative operation can be inverted, or a formula can be rewritten to define some variable by a functional dependency. Also, tuples and tuple sets can be separated into their components, or isolated values can be grouped into a tuple. Finally, simplification operators may be applied on expressions, including the replacement of a $\text{smap}_o\text{filter}$ expression by an equivalent one. For space reasons, such transformation operators are listed in the supplemental material.

We make no claims that these operators are complete, but it is significant that for many constraints it is possible to refine efficient pruning rules using the operators listed above.

5 The Representation of Tuple Variables

We now bring together some existing representations of tuple variables, and propose a notation to combine them, thereby making explicit their design space.

A $(k)$-tuple is a finite sequence $\langle v_1,\ldots,v_k \rangle$ of $k$ elements $v_i$, called components; the tuple has arity $k$. A tuple variable is a variable whose domain is a set of tuples of a given arity.

We consider that the domains of conventional variables are subsets of $\mathbb{Z}$, hence the domain of a $k$-tuple variable is a subset of $\mathbb{Z}^k$. A binary operation $\oplus$ is lifted component-wise from integers to tuples (i.e., $(v_1,\ldots,v_k) \oplus (w_1,\ldots,w_k) = (v_1 \oplus w_1,\ldots,v_k \oplus w_k)$, and pointwise from tuples to sets of tuples (i.e., $S \oplus T = \{s \oplus t \mid s \in S \land t \in T\}$). Two tuples are equal if they are component-wise equal (i.e., $\langle v_1,\ldots,v_k \rangle = \langle w_1,\ldots,w_k \rangle \equiv v_1 = w_1 \land \cdots \land v_k = w_k$).

We denote $\pi_i(T)$ the projection of a $k$-tuple set $T$ onto its $i$th component, with $i \in 1..k$ (i.e., $\{t_i \mid \langle t_1,\ldots,t_k \rangle \in T\}$).

Maintaining extensionally the set of tuples is one way to represent the domain of a tuple variable. Other representations may be used, either because they are sufficient to represent faithfully the domain, or because one is satisfied with the tradeoff between speed and consistency achievable by an over-approximation of the domain. Upon giving some terminology for the well-studied base case of 1-tuple variables, that is conventional integer variables, it suffices here to study 2-tuple variables as an instance of $k$-tuple variables.

A 1-tuple variable has a set of integers as domain. The extensional representation, denoted $\mathcal{E}$, may use a bit vector, a list of intervals, a sparse set, etc. Compact representations take constant space, and all operations on them can be performed in constant time. However, they cannot represent all sets in an exact manner and often over-approximate them. Examples are the interval representation, denoted $\mathcal{I}$, where only the upper and lower bounds of the set are maintained, the i-1pp mode, denoted $\mathcal{Z}$ (Katsirelos, Narodytska, and Walsh 2012), and the congruence representation, denoted $\mathcal{C}$ (Lecointe and Berstel 2006).

A 2-tuple variable has a set of integer pairs as domain. The extensional representation, denoted $2\mathcal{E}$, may use a bit matrix, a list of pairs, etc.

The projection representation possibly over-approximates a pair set $S$ by the Cartesian product $\pi_1(S) \times \pi_2(S)$, where
each of the two projections can in turn use any of the representations of sets. If \( X \) and \( Y \) are representations of 1-tuple variables, then we denote such a representation by \( X \times Y \). For example, if both components are represented by intervals \( I \), then the representation for a 2-tuple variable is denoted by \( I \times I \), or \( I^2 \).

The mapping representation, encountered in Examples 3 and 4, possibly over-approximates a pair set \( S \) by mapping each value of one component to the set of values of the other component (i.e., either every \( v \) in \( \pi_1(S) \) is mapped to the set of values \( \pi_2(S) = \{ w \mid \langle v, w \rangle \in S \} \), or vice-versa). This representation is asymmetric with respect to the two components of the pair: the mapped component must be represented extensionally (\( E \)), while the other component can be represented by any representation \( X \). We denote such a representation by \( E \rightarrow X \). For example, \( E \rightarrow I \) uses intervals.

The polyhedron representation maintains a convex polyhedron, representing a convex envelope of points in a plane. An instance is the octagon representation (Truchet, Pelleau, and Benhamou 2010), denoted \( 2O \) for 2-tuple variables.

The MDD representation, denoted \( 2M \) for 2-tuple variables, adapts the bounded-width multivalued decision diagrams of (Hoda, van Hoeve, and Hooker 2010) to work with only two variables and not the whole set of variables.

The generalisation from sets of \( (k = 2) \) to sets of \( k \)-tuples \( (k > 2) \) is straightforward and omitted here for space reasons. Previous work on \( k \)-tuple variables has considered only the projection representation \( E^k \), for instance (Quimper and Walsh 2005; Michel and Van Hentenryck 2012), and the extensional representation \( kE \), for instance (Bessiere et al. 2008). There are however many other possibilities between those two extremes.

The grammar in Figure 1 summarises the choices of tuple variable representations. Other representations are studied in the field of abstract interpretation, but have not yet been used for constraint programming solvers, to the best of our knowledge. Additional base cases can be added to our grammar as they appear.

**6 Rule Control and Design Methodology**

Before introducing a propagator design methodology enabled by our framework, we briefly discuss three commonly used ways to implement the control of the pruning rules in a constraint programming solver.

A *decomposition* uses several propagators, typically one per constraint in the DP formulation. Their details must in turn be given (but are often left open in the literature).

A single propagator can implement all the pruning rules and apply them in a specific order. In our case, one can apply first a forward phase to prune for the \( A \), tuple link variables, then a backward phase for those variables, and finally prune for all other variables. Two extremes are usually considered. The first is a *non-incremental propagator*, where no data structure is maintained between two calls to the propagator. The second is an *incremental propagator*, for which both the domains and all supports are maintained in the form of a layered graph. This graph is based on ideas from dynamic programming and was first introduced in constraint programming for the Knapsack constraint (Trick 2003).

The control of the propagator includes also the possibility of a preprocessing phase and of the combination of several pruning rules in one. Although this can be described within our framework, and is indeed used for Deviation and SEQBIN, we do not present this here for lack of space.

Our framework enables a methodology for propagator design, composed of the following steps. First, the considered constraint is written as a DP formulation. Then, tuple variables are introduced in the DP formulation to bundle the link variables. Last, in no particular order, a domain representation for the tuple variables is chosen, the pruning rules are refined, and a control of the pruning rules is picked. The three parametrisation axes are almost orthogonal to each other: one can consider each of them in turn and then combine them freely, with the possibility of designing propagators with various properties (see below). This freedom helps one consider useful combinations that might not be considered otherwise. However, it is also possible to consider combinations that make little sense: our framework offers a methodology, but we do not aim at replacing the creativity one can use in the design of propagators. Guiding principles can be stated, but this is beyond the scope of this paper; note for instance that the stepwise refinements of pruning rules in Examples 3 and 4 followed the same abstract recipe.

Further, it is possible to compare tuple variable domain representations and pruning rules under partial orders. For two domain representations \( X \) and \( Y \), we say that \( X \) is stronger than \( Y \) if \( X \) can represent in exact fashion all sets that \( Y \) can represent. For two pruning rules \( r_1 \) and \( r_2 \), we say that \( r_1 \) is stronger than \( r_2 \) if, for each possible domain, \( r_1 \) computes a (not necessarily strict) subset of the set produced by \( r_2 \). Examples are given in Section 7.

**7 The Design of New Propagators**

To show that our framework can be used to design quickly new propagators, we now study the design of propagators for the \( \text{LONGESTPLATEAU}(X_0 \cdots X_n, L) \) constraint, which holds if \( L \) is the length of the longest plateau (sequence of identical elements) within \( X_0 \cdots X_n \). This constraint is called \( \text{LONGEST_CHANGE}(L, X_0 \cdots X_n, =) \).
K (Beldiceanu et al. 2007). A possible DP formulation is:

\[ K_0 = 1 \land M_0 = 1 \quad (C_F) \]

if \( X_i = X_{i-1} \)

then \( K_i = K_{i-1} + 1 \land M_i = M_{i-1} \quad i \in 1..n \quad (C_i) \)

else \( K_i = 1 \land M_i = \max(M_{i-1}, K_{i-1}) \)

\[ L = \max(M_n, K_n) \quad (C_L) \]

where the new variables \( K_i \) and \( M_i \) represent the lengths of the current plateau and the currently longest plateau. The link variables are the \( X_i \), \( K_i \), and \( M_i \) (so \( a = 3 \) and \( b = 0 \)) and we bundle them into 3-tuple variables \( \text{XKM}_i \).

This constraint is useful in rostering, but has no published propagator. It can be handled by SLIDE (with domain representation \( 3E \) for \( \text{XKM}_i \)) and AUTOMATON (with domain representation \( E^2 \)), both implemented by decomposition.

With our framework, we can easily consider other domain representations and controls. We compare the representations \( 3E, E \to E^2, E \to 2O, \) and \( E \to T^2 \). We only use one control, namely a non-incremental propagator. We refine only one pruning rule set, given in the supplemental material; e.g., the rule to prune \( \text{XKM}_i \) based on \( C_i \) is obtained by instantiation from the definition of \( C_i \), functionalisation of \( K_i \) and \( M_i \), embedding and setification of \( K_{i-1} \) and \( M_{i-1} \), and grouping of \( K_{i-1} \) and \( M_{i-1} \) into a pair inside the \( \max \) operator, so as to enable the use of an implementation of \( \max \) that is specific to each representation.

The complexity of a call to each propagator is given in the second column of Table 1, where \( d \) is the size of the largest domain of the \( X_i \), and \( m \) is the maximum value in the domain of \( L \). Those complexities correspond to a direct implementation of the rules. However, it is possible to decrease them by a factor of \( d \) by exploiting the row convexity property, as done for SEQBin by (Petit, Beldiceanu, and Lorca 2011; Katsirelos, Narodytska, and Walsh 2012).

One can establish theoretically, for any 3-tuple variable, that \( 3E \) is stronger than all other representations, and that \( E \to E^2 \) and \( E \to 2O \) are incomparable but stronger than \( E \to T^2 \). Experimentally, for LONGESTPLATEAU, we compare the pruning strengths (Independently of search) of these representations with the best that can be achieved without tuple variables, corresponding to ensuring domain consistency (DC) of each \( C_i \) with an \( E^2 \) representation. For several illustrative combinations of \( n \) and \( d \), we randomly sample all possible domains for the \( X_i \) and \( L \) variables. The results are reported in Table 1, by giving the average reduction of the product of the domain sizes with respect to the maximum possible reduction (obtained by global domain consistency). We only consider instances where some pruning is possible. The last line of Table 1 reports the percentage of additional instances that were generated to produce 10,000 prunable ones. The penultimate line reports the percentage of infeasible instances. All propagators, except the one without tuple variables, were able to recognize all the infeasible instances.

From Table 1, it is clear that using tuple variables is very beneficial. The \( E \to 2O \) representation seems to present the best compromise between time complexity and pruning on this random sampling. On a structured benchmark, the interaction between pruning strength and search is complicated and orthogonal to the purpose of this section: the important point here is that, using our framework, it is easy to design several propagators and compare their performance.

### Table 1: Domain representations for LONGESTPLATEAU

<table>
<thead>
<tr>
<th>Domains</th>
<th>Complexity</th>
<th>Pruning strength (( n \cdot d ); in %)</th>
<th>5-2</th>
<th>5-30</th>
<th>20-2</th>
<th>20-30</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 3E )</td>
<td>( O(nd^2) )</td>
<td>100 100 100 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E \to E^2 )</td>
<td>( O(nd^2) )</td>
<td>90 52 88 89</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E \to 2O )</td>
<td>( O(nd^2) )</td>
<td>94 68 86 90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E \to T^2 )</td>
<td>( O(nd^2) )</td>
<td>85 52 81 89</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC without tuple var.</td>
<td>( O )</td>
<td>62 21 45 53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infeasible inst. (in %)</td>
<td>( O )</td>
<td>21 10 20 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>additional inst. (in %)</td>
<td>( O )</td>
<td>35 117 37 126</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(8 Conclusion and Future Work)

We presented a framework for designing propagators operating on variable sequences. Many published propagators are indeed close enough to each other that they can be described as instances of a generic propagator, based on dynamic programming principles. In particular, we showed that one can describe very concisely the pruning rules and the representation of the introduced tuple variables. We also showed how one can refine these pruning rules from a formulation of the constraint using a few transformation operators.

As there are many choices for the pruning rules, the representation of tuple variables, and the control of a propagator, our framework presents several advantages. It offers a common language for describing the differences and commonalities between propagators. It is possible to explore conceptually and systematically the alternative choices when deriving a propagator. The implementation of propagators is simplified: the solver-specific code for tuple variable representations and pruning rule control can be shared among many constraints; once those have been written, the implementation phase is reduced to the translation of the pruning rules into solver-specific code.

Our framework is conceptual and solver independent. A tool for our framework would make it possible to (semi-)automate the development of propagators. One could then generate alternative propagators and benchmark them, as done, e.g., by (Akgun et al. 2013) for choosing among alternative models of a class of problems. Even without tool support, our framework enables a methodology of propagator design; this is useful, as illustrated in Section 7.

The proposed framework can benefit from many improvements. While we offered a language for the representations of the tuple variables and pruning rules, there is still a lot to do to achieve a more precise characterisation of the control of a propagator, as done, e.g., by (Régis 2005) for arc consistency algorithms. Properties, such as the time and space complexities, can be computed from the description of the designed propagators. Other properties, such as idempotency and the achievement of domain consistency, cannot be inferred from such a description and often require complex proofs anyway. Properties of the considered constraint, such as monotonicity or row convexity, can also be exploited.
Acknowledgements

This work is supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council (VR). We thank the reviewers for their constructive comments.

References


Leconte, M., and Berstel, B. 2006. Extending a CP solver with congruences as domains for program verification. In CP Workshop on Software Testing, Verification and Analysis, 22–33.