EPiC Series in Computer Science

Volume 36, 2015, Pages 113–126

GCAI 2015. Global Conference on Artificial Intelligence

Implied Constraints for AUTOMATON Constraints

María Andreína Francisco Rodríguez, Pierre Flener, and Justin Pearson

Uppsala University, Department of Information Technology, SE – 751 05 Uppsala, Sweden
{Maria.Andreina.Francisco, Pierre.Flener, Justin.Pearson}@it.uu.se

Abstract

Automata allow many constraints on sequences of variables to be specified in a high-level way for constraint programming solvers. An automaton with accumulators induces a decomposition of the specified constraint into a conjunction of constraints with existing inference algorithms, called propagators. Towards improving propagation, we design a fully automated tool that selects, in an off-line process, constraints that are implied by such a decomposition. We show that a suitable problem-specific choice among the tool-selected implied constraints can considerably improve solving time and propagation, both on a decomposition in isolation and on entire constraint problems containing the decomposition.

1 Introduction

In constraint programming (CP) [20], frameworks are given in [2, 7, 18] for specifying a constraint on a sequence of variables in a high-level way by means of a finite automaton, possibly augmented with accumulators in the framework of [7]. An automaton can be seen as a *checker* for ground instances of the specified constraint. For example, in a nonogram puzzle, a row constrained to contain two stretches of black cells, of lengths 4 and 3 in this order, separated by at least one white cell but preceded and followed by any amounts of white cells, can be checked by an automaton equivalent to the regular expression $\mathrm{w}^*\mathrm{b}^4\mathrm{w}^+\mathrm{b}^3\mathrm{w}^*$, where the row is represented by an array of variables, whose domain value 'w' stands for white and 'b' for black.

The framework of [18] lifts an automaton without accumulators into a *propagator* for the specified constraint. It maintains hyper-arc consistency [20], here called *domain consistency*: all infeasible values are removed from the domains of all variables. The more general framework of [7] lifts an automaton with accumulators into a *decomposition* of the specified constraint in terms of constraints with existing propagators (we will see the details in Section 2.3).

In this paper, we focus on the more general framework of [7], where accumulators were motivated by the need to specify a constraint C on a sequence X of variables using an automaton whose size does not depend on the length of X: accumulators are initialised at the start state and evolve through the transitions; upon acceptance, the accumulators are usually linked to a result variable of C via an arithmetic constraint. The *Global Constraint Catalogue* [4] gives very compact automata with accumulators and at most five states for 59 constraints (and some will be given in Section 2). However, maintaining domain consistency for AUTOMATON is in general NP-hard [9]. Getting close to domain consistency is the challenge we tackle in this paper.

We continue our earlier work [17], where we added constraints implied by the decomposition in order to improve propagation: we *manually* translated an automaton with accumulators into an imperative checker, with a loop iterating over the input symbols, fed the checker into an off-the-shelf automated loop-invariant generator, and *manually* translated each loop invariant into an implied constraint. We did so for *two* particular constraints, but we proved for one of them that domain consistency is maintained at no asymptotic overhead for the decomposition extended by a manual choice of implied constraints. We experimentally studied the impact of the implied constraints only on the decompositions, but *not* on entire constraint problems.

After a summary of the background material in Section 2, enabling us to make a very precise formulation of our objective in Section 2.4, the **contributions** and **impact** of Sections 3 and 4 of this paper are as follows:

- We design a *fully automated* tool (at http://www.it.uu.se/research/group/astra/software/impGen.zip) that reads *any* automaton, in the SICStus Prolog syntax, of a large subclass of those in the Global Constraint Catalogue and selects a set of constraints that are implied by the decomposition of the constraint specified by the automaton.
- Our tool is *not* based on an off-the-shelf loop-invariant generator. For better control, we design our own generator, which works *directly* on the automaton (rather than on its translation into an imperative program) and *directly* generates implied constraints (rather than loop invariants that have to be translated into implied constraints). Our tool is based on Farkas' lemma and has parameters for controlling the quality of its results.
- Our tool eliminates uninteresting and propagation-redundant constraints from the generated set of implied constraints, so as to ease the user's choice of implied constraints that are actually added to the decomposition. The latter choice is *problem*-specific and beyond the scope of our tool.
- We experimentally show that a suitable choice of implied constraints can improve propagation on the *decomposition*. We do not aim at maintaining domain consistency for the decomposition. On entire constraint *problems*, solving time can be reduced despite the overhead in running more propagators for the decomposition.
- Since our implied constraints are *linear*, we strongly believe they are also relevant in the context of integer-programming decompositions, such as in [16] for REGULAR, of constraints specified by automata with accumulators: first experiments [1] confirm this.

In Section 5, we conclude and discuss other related work as well as future work.

2 Background: Constraints on Automata

We define background concepts that are not commonly known. We add running examples for the rest of this paper, and precisely state its objective.

2.1 Automata, the REGULAR and AUTOMATON Constraints

Recall that a *deterministic finite automaton* (DFA) is a tuple $\langle Q, \Sigma, \delta, q_0, A\rangle$, where Q is the set of *states*, Σ the *alphabet*, $\delta\colon Q \times \Sigma \to Q$ the *transition function*, $q_0 \in Q$ the *start state*, and $A \subseteq Q$ the set of *accepting states*. When $\delta(q, \sigma) = q'$, there is a transition from state q to state q' upon consuming alphabet symbol σ in the word given to the DFA. Let Σ^* denote the infinite set of words built from Σ, including the empty word, denoted ϵ. The *extended transition function* $\widehat{\delta}\colon Q \times \Sigma^* \to Q$ for words (instead of symbols) is recursively defined by $\widehat{\delta}(q, \epsilon) = q$ and $\widehat{\delta}(q, w\sigma) = \delta(\widehat{\delta}(q, w), \sigma)$ for a word w and symbol σ. Note that δ and $\widehat{\delta}$ are total functions.

A word w is *accepted* if $\widehat{\delta}(q_0, w) \in A$. An example will be given soon, but first we argue for augmenting DFAs with a memory, in the spirit of [7], to encode compactly many constraints.

A DFA is useful for encoding a constraint on a sequence X of variables: the $\textsc{Regular}(\mathcal{D}, X)$ constraint [18] holds if the word represented by X is accepted by DFA $\mathcal{D}$. There are propagators achieving domain consistency for Regular in time polynomial in the DFA size [18]. DFAs augmented with a memory [7] were motivated by the need to encode a constraint on a sequence X using an automaton whose size does not depend on the length of X.

Example 1. In a sequence, a *group* [4] is a maximal contiguous subsequence with values from a given set. The $\textsc{nGroup}(X, W, N)$ constraint [4] holds if there are N groups of values from the given set W in the possibly empty sequence X of variables. The ground instance $\textsc{nGroup}([\mathrm{a}, \mathrm{d}, \mathrm{d}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b}], \{\mathrm{a}, \mathrm{e}\}, 2)$ holds since there are 2 groups of occurrences of ‘a’ and ‘e’ in the sequence $[\mathrm{a}, \mathrm{d}, \mathrm{d}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b}]$, namely $[\mathrm{a}]$ and $[\mathrm{e}, \mathrm{a}]$. This constraint of the CHIP solver is useful in staff rostering, where counting constraints on a sequence (the shift assignments of an employee over a planning horizon, say) are frequent. It has no known propagator, and encoding it using Regular requires designing a DFA whose size depends on the length of X. □

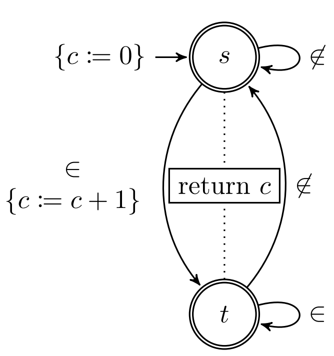
We here define a *memory-DFA* (mDFA) with a memory of $k \geq 0$ integer accumulators as a tuple $\langle Q, \Sigma, \delta, q_0, I, A, \alpha \rangle$, where Q, Σ, q_0, and A are as in a DFA, while the transition function δ has signature $(Q \times \mathbb{Z}^k) \times \Sigma \to Q \times \mathbb{Z}^k$, and similarly for its extended version $\widehat{\delta}$. Further, I is the k-tuple of initial values of the accumulators in the memory. Finally, $\alpha \colon A \times \mathbb{Z}^k \to \mathbb{Z}$ is a total function called the *acceptance function* and transforms the memory of an accepting state into an integer. Given a word w, the mDFA returns $\alpha(\widehat{\delta}(\langle q_0, I \rangle, w))$ if w is accepted.

Example 2. Consider the mDFA $\mathcal{N}$ in Figure 1a. It returns the number of groups of ‘$\in$’ within a word over the alphabet $\Sigma = \{\in, \notin\}$. It uses $k = 1$ accumulator: at any moment, accumulator c stores the number of groups seen so far. The state set Q is $\{s, t\}$. The start state q_0 is s, and is indicated by an arrow coming from nowhere, annotated within braces by the initialisation to zero of c, hence $I = \langle 0 \rangle$. A transition $\delta(\langle q, \langle c \rangle \rangle, \sigma) = \langle q', \langle c' \rangle \rangle$, where c' is a functional expression in terms of c, is depicted by an arrow going from state q to state q', annotated by symbol σ and, within braces, the memory update $\langle c \rangle := \langle c' \rangle$. For instance, the arc from s to t depicts that $\delta(\langle s, \langle c \rangle \rangle, \in) = \langle t, \langle c+1 \rangle \rangle$ for all c. If an update corresponds to the identity function, then we do not depict it; for instance, both self-loops have no depicted updates, as $\langle c \rangle := \langle c \rangle$. All states are accepting, hence $A = Q$. The acceptance function α transforms a memory $\langle c \rangle$ at both states into c, and is depicted by a box linked to both states by a dotted line. Note how the size of $\mathcal{N}$ does not depend on the length of the word it consumes. □

The $\textsc{Automaton}(\mathcal{M}, X, R)$ constraint [7] holds if the word represented by the sequence X of variables is accepted by mDFA $\mathcal{M}$ and variable R is the integer returned by $\mathcal{M}$, that is $R = \alpha(\widehat{\delta}(\langle q_0, I \rangle, X))$. If $\mathcal{M}$ has $k = 0$ accumulators (and hence there are no α and R), then this constraint specialises to $\textsc{Regular}(\mathcal{M}, X)$. The feasibility test of Automaton is NP-hard [9]. The decomposition of [7] will be given in Section 2.3, but we first need to make an observation.

2.2 Signature Variables and Signature Constraints

A constraint C on a sequence X of variables can often be encoded with the help of a DFA or mDFA that operates not on X, but on a sequence S of variables that are called *signature variables*, each depending via a *signature constraint* [7] on a sliding window of a consecutive variables within X. The constant $a \geq 1$ is called the *arity* of the signature constraints.



(a)

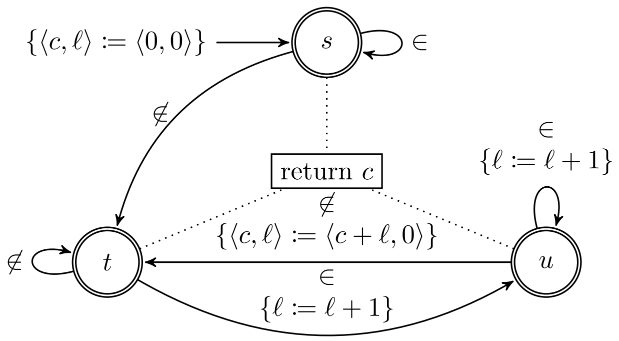

$\{\langle c,\ell\rangle \coloneqq \langle 0,0\rangle\}$
s
$\in$
$\notin$
return c
$\in$
$\{\ell \coloneqq \ell+1\}$
$\notin$
$\{\langle c,\ell\rangle \coloneqq \langle c+\ell,0\rangle\}$
$\notin$
t
u
$\in$
$\{\ell \coloneqq \ell+1\}$
(b)

Figure 1: (a) Memory-DFA $\mathcal{N}$ with one accumulator for nGROUP(X, W, N).
(b) Memory-DFA $\mathcal{F}$ with two accumulators for FULLGROUPNVAL(X, W, T).

Example 3. Consider the nGROUP$([X_1, \ldots, X_n], W, N)$ constraint of Example 1. We constrain a sequence $[S_1, \ldots, S_m]$ of $m = n$ signature variables S_i with domain $\{\in, \notin\}$ by the signature constraints $(X_i \in W \Leftrightarrow S_i = `\in') \wedge (X_i \notin W \Leftrightarrow S_i = `\notin')$ for all $1 \leq i \leq n$: we have $a = 1$ since each signature constraint is on a single X_i. Using the mDFA $\mathcal{N}$ of Example 2 and Figure 1a we encode nGROUP$([X_1, \ldots, X_n], W, N)$ by AUTOMATON$(\mathcal{N}, [S_1, \ldots, S_n], N)$ and these signature constraints. For the ground instance nGROUP$([\mathrm{a}, \mathrm{d}, \mathrm{d}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b}], \{\mathrm{a}, \mathrm{e}\}, 2)$, the mDFA $\mathcal{N}$ returns $c = 2$ on the signature sequence $[\in, \notin, \notin, \notin, \in, \in, \notin]$. □

Example 4. In a sequence, a *full group* [4] is a group that does not include the first or last element of the sequence. The FULLGROUPNVAL(X, W, T) constraint [4] holds if the full groups for the set W in the possibly empty sequence X of variables have T elements in total. The ground instance FULLGROUPNVAL$([\mathrm{a}, \mathrm{d}, \mathrm{d}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b}], \{\mathrm{a}, \mathrm{e}\}, 2)$ holds since there is one full group of occurrences of 'a' and 'e' in the sequence $[\mathrm{a}, \mathrm{d}, \mathrm{d}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b}]$, namely $[\mathrm{e}, \mathrm{a}]$, with 2 elements, but not the group $[\mathrm{a}]$. Using the mDFA $\mathcal{F}$ of Figure 1b, we encode FULLGROUPNVAL$([X_1, \ldots, X_n], W, T)$ by AUTOMATON$(\mathcal{F}, [S_1, \ldots, S_n], T)$ and the signature constraints of Example 3. Note that accumulator ℓ, which denotes the length of the current group, is *reset* to zero in the transition from state u to state t, when the previous group is detected to have been a full group. □

Example 5. In an integer sequence $[X_1, \ldots, X_n]$, an *inflexion* [10] is a maximal contiguous subsequence $[X_i, \ldots, X_j]$ with $X_i \leq X_{i+1} \leq \cdots \leq X_{j-1} > X_j$ or $X_i \geq X_{i+1} \geq \cdots \geq X_{j-1} < X_j$, where $j \geq i + 2$. The INFLEXION(X, N) constraint [10] holds if there are N inflexions in the non-empty sequence X of integer variables. The ground instance INFLEXION$([1, 1, 4, 8, 2, 7, 1], 3)$ holds since there are 3 inflexions in the sequence $[1, 1, 4, 8, 2, 7, 1]$, namely $[1, 1, 4, 8, 2]$, $[8, 2, 7]$, and $[2, 7, 1]$. Using the mDFA $\mathcal{I}$ of Figure 2, we encode INFLEXION$([X_1, \ldots, X_n], N)$ by AUTOMATON$(\mathcal{I}, [S_1, \ldots, S_{n-1}], N)$ and the signature constraints $(X_i < X_{i+1} \Leftrightarrow S_i = `<') \wedge (X_i = X_{i+1} \Leftrightarrow S_i = `=') \wedge (X_i > X_{i+1} \Leftrightarrow S_i = `>')$ for all $1 \leq i < n$, with arity $a = 2$. □

The Global Constraint Catalogue gives very compact memory-DFAs, with accumulators and at most 5 states, for currently 59 constraints.

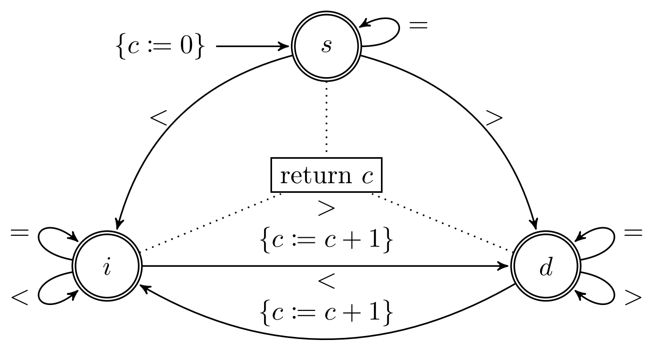


Figure 2: Memory-DFA $\mathcal{I}$ with one accumulator for Inflexion(X, N).

2.3 Decomposition of the Automaton Constraint

Consider a constraint $C([X_1, \ldots, X_n], R)$ encoded by an Automaton$(\mathcal{M}, [S_1, \ldots, S_m], R)$ constraint and signature constraints channelling between the variables X_i and the signature variables S_i. In the absence of signature constraints and signature variables, we consider $S_1 = X_1 \wedge S_2 = X_2 \wedge \cdots \wedge S_m = X_n$, with $m = n$, to be the signature constraints, as this simplifies the discussion. Let the mDFA $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, I, A, \alpha \rangle$ have k accumulators. The Automaton$(\mathcal{M}, [S_1, \ldots, S_m], R)$ constraint has the following decomposition [7, 3]:

$$\begin{array}{c} Q^0 = q_0 \wedge \langle C_1^0, \ldots, C_k^0 \rangle = I \wedge Q^m \in A \wedge \alpha(\langle C_1^m, \ldots, C_k^m \rangle) = R \wedge \\ \bigwedge_{i=1}^m \textsc{Trans}(Q^{i-1}, \langle C_1^{i-1}, \ldots, C_k^{i-1} \rangle, S_i, Q^i, \langle C_1^i, \ldots, C_k^i \rangle) \end{array} \tag{1}$$

where:

- Each Q^i is a new variable, called a *state variable*, with domain Q: it denotes the state of $\mathcal{M}$ after the values of the signature variables $S_1, \ldots, S_i$ have been consumed, with $0 \leq i \leq m$.
- Each C_j^i is a new integer variable, called an *accumulator variable*: it denotes the value of the j^{th} accumulator of $\mathcal{M}$ after $S_1, \ldots, S_i$ have been consumed, with $0 \leq i \leq m$.
- The ground instance $\textsc{Trans}(q, \langle c_1, \ldots, c_k \rangle, \sigma, q', \langle c'_1, \ldots, c'_k \rangle)$ holds if δ transits from state q to state q' for symbol $\sigma \in \Sigma$ and updates the tuple of k accumulators from the values $\langle c_1, \ldots, c_k \rangle$ to the arithmetic expressions $\langle c'_1, \ldots, c'_k \rangle$; it is called a *transition constraint.*

The Automaton constraint of SICStus Prolog [15] is implemented as (1), with Trans encoded via Table and arithmetic constraints for the accumulator updates. Even if domain consistency were maintainable efficiently for Trans, maintaining domain consistency for Automaton is in general NP-hard [9]: encoding the potential accumulator values in the automaton states to enjoy the polynomial-time domain consistency of Regular leads to combinatorial explosion.

2.4 Precise Statement of Our Objective

We are now able to state precisely our objective. We aim at automatically generating implied constraints that can improve propagation for the decomposition of an Automaton$(\mathcal{M}, S, R)$ constraint where $\mathcal{M}$ has at least one accumulator. The generation is specific to $\mathcal{M}$ but not to S and is done off-line. It is very important to understand that we do not aim at maintaining domain consistency for the decomposition extended by the implied constraints: our tool cannot make any such guarantees. If one *experimentally* observes that domain consistency is achieved for all *tried* instances in the presence of implied constraints, then one has to make a custom *proof* that domain consistency is actually maintained *in general* in their presence: such a proof can be very involved, witness [17, section III.C] for a constraint not discussed here.

We focus on memory-DFAs where every accumulator update is a *linear* expression on the accumulators. This includes increments and decrements by constant amounts (as in $c \coloneqq c+1$) or by other accumulators (as in $c \coloneqq c+\ell$), resets (as in Example 4), etc. This excludes updates using the 'max' and 'min' operators, for instance. We will revisit this issue in the conclusion. In the Global Constraint Catalogue, 12 of the mentioned 59 constraints with memory-DFAs are within our scope, including those of Examples 3 to 5, but hundreds more are given in [5].

Further, we focus on implied constraints that are *linear* inequalities on the accumulator and state variables (upon numbering the states of $\mathcal{M}$). We will revisit this issue in the conclusion.

3 Generation of Linear Implied Constraints

Our approach to inferring constraints implied by the decomposition of an AUTOMATON$(\mathcal{M}, S, R)$ constraint consists of three steps. First, using one half of Farkas' lemma and a linear template T for implied constraints, we set up a system N of non-linear constraints that model T being true at every state of the memory-DFA $\mathcal{M}$ (Section 3.1). Second, we solve N, each solution providing an instantiation of T into a particular linear implied constraint (Section 3.2). Third, we eliminate uninteresting and *propagation*-redundant constraints from the generated set of implied constraints (Section 3.3). We consider the user's choice of implied constraints that are actually added to the decomposition to be a problem-specific rather than constraint-specific task, so our tool offers no help in that choice, as it focusses on *suggesting* implied constraints.

3.1 Implied Constraints: Template and Set-Up of the System N

We adapt the recipe of [21] for linear transition systems. A *linear transition system* does not have notions of consumption and acceptance of words, but is otherwise like a memory-DFA if every accumulator update of the latter is a linear expression on the accumulators. Everything that follows requires linearity, also of the implied constraints, so we now make that restriction.

One half of Farkas' lemma (e.g., [14]) says that a system of e linear inequalities $a_{i1}y_1 + \cdots + a_{ik}y_k + b_i \geq 0$ over k real-valued variables y_j has another linear inequality $\alpha_1 y_1 + \cdots + \alpha_k y_k + \beta \geq 0$ over the same variables as a logical consequence if the latter is equal to a linear combination of the former, that is, if there exist e real numbers $\lambda_i \geq 0$ such that $\alpha_j = \sum_{i=1}^{e} \lambda_i a_{ij}$, for $1 \leq j \leq k$, and $\beta \geq \sum_{i=1}^{e} \lambda_i b_i$. The following representation helps to see this:

$$\begin{array}{c|c}
\lambda_1 & a_{11}y_1 + \cdots + a_{1k}y_k + b_1 \geq 0 \\
\vdots & \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
\lambda_e & a_{e1}y_1 + \cdots + a_{ek}y_k + b_e \geq 0 \\
\hline
 & \alpha_1 y_1 + \cdots + \alpha_k y_k + \beta \geq 0
\end{array}$$

If the ith linear constraint is an equality, then the requirement $\lambda_i \geq 0$ is dropped. The other half of Farkas' lemma gives a necessary and sufficient condition for a linear inequality to be a logical consequence of a system of linear inequalities. We do not need it as we do not aim at completeness and thus need not prove that a set of generated implied constraints is complete.

Let variable y_j denote the jth accumulator of $\mathcal{M}$, with $1 \leq j \leq k$. Our linear template T for implied constraints for now is $\alpha_1 y_1 + \cdots + \alpha_k y_k + \beta \geq 0$, where the Greek letters denote the variables for which we will solve constraints. An instance of template T is true at every state of $\mathcal{M}$ if it is true at the start state of $\mathcal{M}$ and if its truth is preserved by every transition of $\mathcal{M}$. We now show how to encode this using Farkas' lemma. For the start state, we encode using Farkas' lemma that the point-wise initialisation equalities behind $\langle y_1, \dots, y_k \rangle = I$ have T as a logical consequence, where I is the k-tuple of initial values of the accumulators of $\mathcal{M}$.

Example 6. Recall the mDFA in Figure 1b for the FULLGROUPNVAL constraint of Example 4. There are $k = 2$ accumulators, called c and ℓ, both initialised to 0. So the template for implied constraints is $\alpha_1 c + \alpha_2 \ell + \beta \geq 0$ and it must be a logical consequence of the $e = k$ initialisation equalities $c = 0$ and $\ell = 0$, with $a_{11} = 1 = a_{22}$, $a_{12} = 0 = a_{21}$, and $b_1 = 0 = b_2$. Using Farkas' lemma, we get the constraints $\exists \lambda_1, \lambda_2 : \alpha_1 = \lambda_1 \wedge \alpha_2 = \lambda_2 \wedge \beta \geq 0$ of N. Some instances of the template for implied constraints that fulfil those constraints are $c \geq 0$ and $c + 2\ell + 1 \geq 0$. $\square$

For each transition $\delta(\langle q, \langle y_1, \dots, y_k \rangle \rangle, \sigma) = \langle q', \langle y'_1, \dots, y'_k \rangle \rangle$ of $\mathcal{M}$, where each y'_j is a linear functional expression in terms of all the y_j, we encode using Farkas' lemma that template T has $T[y/y']$ as a template logical consequence, where $T[y/y']$ denotes T with every y_j substituted by y'_j. The resulting constraints are in general non-linear.

Example 7. Continuing from Example 6, first consider the two transitions to state u, upon which accumulator ℓ is incremented by 1. The desired template logical consequence $T[y/y']$ of T is $\alpha_1 c + \alpha_2(\ell + 1) + \beta \geq 0$. Using Farkas' lemma and rearranging, we get for each of these transitions the non-linear constraints $\exists \lambda_3 \geq 0 : \alpha_1 = \lambda_3 \alpha_1 \wedge \alpha_2 = \lambda_3 \alpha_2 \wedge \alpha_2 + \beta \geq \lambda_3 \beta$ of N. Second, consider the transition from u to t, upon which accumulator c is incremented by ℓ and accumulator ℓ is then reset to zero. The desired template logical consequence $T[y/y']$ of T is $\alpha_1(c + \ell) + \alpha_2 0 + \beta \geq 0$. Using Farkas' lemma and rearranging, we get the non-linear constraints $\exists \lambda_4 \geq 0 : \alpha_1 = \lambda_4 \alpha_1 \wedge \alpha_1 = \lambda_4 \alpha_2 \wedge \beta \geq \lambda_4 \beta$ of N. Finally, for the self-loop on state t, upon which there is no accumulator update, we get the non-linear constraints $\exists \lambda_5 \geq 0 : \alpha_1 = \lambda_5 \alpha_1 \wedge \alpha_2 = \lambda_5 \alpha_2 \wedge \beta \geq \lambda_5 \beta$ of N. $\square$

We now go beyond adapting the recipe of [21], by discussing three refinements of the ideas seen so far. First, many implied constraints that provide extra propagation are expressed not only on the *current* values of the accumulators, but also on their values upon *previous* transitions. For example, only implied constraints that provide no extra propagation, such as $c \geq 0$ and $\ell \geq 0$, result from the solutions to the constraint system N we have set up in Examples 6 and 7: those examples were simple enough to explain all features of the procedure, but simpler than practical applications thereof. The following other example is enlightening.

Example 8. Recall the mDFA in Figure 1a for the NGROUP constraint of Example 1: it has one accumulator, called c, and $c \leq c_2 + 1$ does provide extra propagation [17], where the new accumulator c_2 denotes the value of c two transitions ago. We say that the *history length* is 2. Let another new accumulator c_1 denote the value of c one transition ago. Upon adding the initialisation $\langle c_2, c_1 \rangle \coloneqq \langle 0, 0 \rangle$ to the start state, and adding the accumulator update $\langle c_2, c_1 \rangle \coloneqq \langle c_1, c \rangle$ to each transition, we get the template $\alpha_1 c_2 + \alpha_2 c_1 + \alpha_3 c + \beta \geq 0$, so that the desired implied constraint $c \leq c_2 + 1$ corresponds to $\alpha_1 = 1 = \beta \wedge \alpha_2 = 0 \wedge \alpha_3 = -1$. $\square$

Our tool allows the user to indicate the history length h, so that appropriate accumulator terms are added to the template T. Things scale when $h \cdot k$ grows, since the process is off-line.

Second, the template T can be extended by adding a term ρq for the state q at which the automaton is. This requires numbering the states. For example, this extension is actually necessary for generating the implied constraint $c \leq c_2 + 1$ of Example 8, as we will see in Example 9. Our tool allows the user to switch on this option.

Third, we have so far described how to generate implied constraints that are true at *every state* of $\mathcal{M}$. We can also make as many copies of the template as there are states in $\mathcal{M}$, so as to aim at generating *state-specific* implied constraints. We must then use the appropriate template copies each time we apply Farkas' lemma for the start state or a transition. For example, an implied constraint specific to the start state s of the mDFA in Figure 1a for the nGroup constraint is $c_1 = c$, inferred from the invariants $c_1 \leq c$ and $c_1 \geq c$; we will see in Example 9 an encoding of this implied constraint when a term on the state variable q is added to the template. Our tool allows the user to switch on this option.

3.2 Implied Constraints: Generation by Solving the System N

So far, we have shown how to set up a system N of non-linear constraints that are on the variables denoted by Greek letters in the template $\alpha_1 y_1 + \cdots + \alpha_k y_k + \rho q + \beta \geq 0$ for implied constraints, but not on its accumulator variables y_j and state variable q. We now show how to solve N so that each solution provides an instantiation of the variables α_j, ρ, and β of the template, yielding an implied constraint on the accumulator variables y_j and state variable q.

Memory-DFAs are defined for integer accumulators, so we solve the non-linear constraint system N for *integer* values of the variables α_j, ρ, β, and λ_i: this is our second deliberate relaxation of completeness. Further, we reckon that for each variable a small finite integer interval centred on zero, such as from -5 to $+5$, suffices for finding many useful implied constraints: this is our last deliberate relaxation of completeness. Since our tool is written in SICStus Prolog and reads automata in the SICStus Prolog syntax used in the Global Constraint Catalogue, we use the *finite*-domain CP solver of SICStus Prolog [15], although we could have used any integer programming solver: we solve upon linearising N by branching on values for the λ_i.

Many implied constraints that provide extra propagation are not generated in one go, even if all options are switched on.

Example 9. Consider the implied constraint $c \leq c_2 + 1$ of Example 8. Let us number state s of the mDFA in Figure 1a as 0 and state t as 1. Generating this implied constraint requires the prior knowledge that $c - c_1 \leq q$, meaning that c and c_1 are equal at the start state s and apart by at most one unit at state t. It turns out that $c - c_1 \leq q$ actually *is* an implied constraint. So let us add this implied constraint to the top side of each application of Farkas' lemma, with its own multiplier λ_p, and set up a second non-linear system N_2. It turns out that $c \leq c_2 + 1$ is now an implied constraint, generated from a solution to N_2. □

Our tool allows the user to indicate an upper bound u on the number of non-linear systems it will set up and solve; it will finish earlier if no new implied constraints are generated at some iteration. Recall that the whole generation process is specific to an automaton but not to the constrained sequence, so that it is off-line and can take an arbitrary amount of time. Our tool takes from seconds to days, depending on the parameters, especially u and the history length h.

3.3 Implied Constraints: Redundancy Elimination and Selection

Some generated implied constraints are useless. For example, when ρ and all the α_j are zero, we can get an implied constraint like $5 \geq 0$, which is vacuously true and cannot improve propagation, but will slow it down. Other generated implied constraints are propagation-redundant. For example, the implied constraint $c \leq c_2 + 1$ of Example 9 is redundant with $3c \leq 3c_2 + 3$, and the former will give better propagation than the latter. As another example, the implied constraint $c + \ell \geq 0$ is redundant with the implied constraints $c \geq 0$ and $\ell \geq 0$ that result from the solutions to the constraint system N we have set up in Examples 6 and 7. Such redundancies stem from our not finding *generators* for the solutions to N.

Our tool automatically eliminates useless and redundant constraints. The remaining implied constraints are suggested to the user, who can experimentally choose a suitable problem-specific subset that provides a good speed-up for the problem at hand.

As the decomposition (1) of $\text{AUTOMATON}(\mathcal{M}, [S_1, \dots, S_m], R)$ reveals accumulator and state variables C_j^i and Q^i for every prefix $[S_1, \dots, S_i]$, with $0 \leq i \leq m$, we post an implied constraint $\alpha_1 y_1 + \dots + \alpha_k y_k + \rho q + \beta \geq 0$ as $\alpha_1 C_1^i + \dots + \alpha_k C_k^i + \rho Q^i + \beta \geq 0$ for every $0 \leq i \leq m$ rather than just for $i = m$. An implied constraint γ specific to state q is posted as $(Q^i = q) \Rightarrow \gamma$.

4 Results

To assess the generated implied constraints, we experimented on decompositions in isolation (Section 4.1) and in the context of entire constrained optimisation problems (Section 4.2). These experiments were run for the constraints of Examples 3 to 5, for which no dedicated propagators are known. Our tool may select hundreds of implied constraints and some of them may not improve propagation. So we first tested individually all the implied constraints that were selected by our tool in less than a minute for $u = 3$, and then chose a small subset of those that provided the greater time or failure reduction on the decomposition in isolation: $c \geq c_2$ and $c_2 + 1 \geq c$ for NGROUP ($h = 2$), $c \geq c_1$ and $c_4 + \ell_4 + 4 \geq c + \ell$ for FULLGROUPNVAL ($h = 4$), as well as $c_1 + 1 \geq c$ and $c_2 + 2 \geq c$ for INFLEXION ($h = 2$). Note that, in the presence of the signature constraints, generating random initial domains for the signature variables $[S_1, \dots, S_m]$ is equivalent to generating random initial domains for the variables $[X_1, \dots, X_n]$. All experiments were run in SICStus Prolog 4.2 [15] on a quad core 3.07 GHz Intel Core i7-950 machine with 8 MB cache, running openSUSE 13.1.

4.1 Experiments on Decompositions in Isolation

We generated instances with sequences $[S_1, \dots, S_m]$ of m signature variables as well as random initial domains for the result variable R (one value, two values, and intervals of length 2 or 3) and the S_i (one or two values for NGROUP and FULLGROUPNVAL, where the signature constraints have arity $a = 2$; one to three values for INFLEXION, where $a = 3$). Unless otherwise indicated, the instances for NGROUP have sequences of length $m = 100$, those for FULLGROUPNVAL have $m = 50$, and those for INFLEXION have $m = 15$: the length m varies for each constraint in order to make the sequences long enough so that it takes at least 0.01 seconds to find all solutions, if any, to most instances. We generated a set of 1,500 satisfiable instances and a set of 1,500 unsatisfiable instances for each constraint in order to show that the implied constraints have different effects on each of the sets. For each constraint, the decomposition alone and the decomposition with implied constraints are tested on the same two sets of instances. The default search strategy is used: leftmost variable first (R before the S_i), lower value first. However, our purpose is orthogonal to picking a search strategy: the implied constraints do not introduce new variables and would thus not affect many search strategies, and we obtained similar results with other search strategies, as we will show in Section 4.2.

The results on the sets of satisfiable instances are shown in Figure 3. The decomposition of NGROUP (left column) with the implied constraints is almost always faster than the decomposition alone, is about 30% faster on average, and has about 60% fewer failures on average. The decomposition of FULLGROUPNVAL (centre column) with the implied constraints is sometimes faster than the decomposition alone, but is about 20% slower on average, and has about 50%

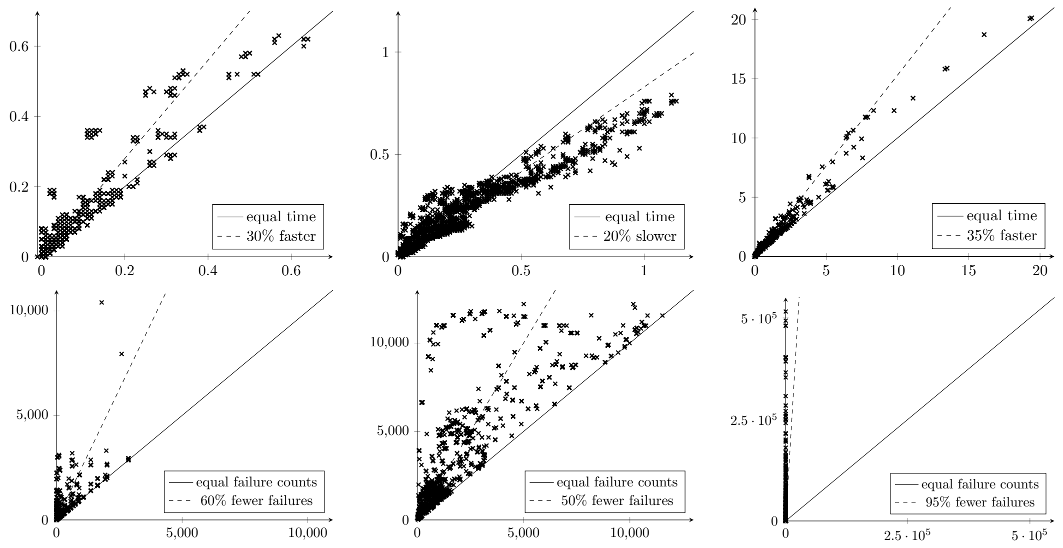


Figure 3: Seconds (top row) and failures (bottom row) to find all solutions to satisfiable instances of NGROUP (left column), FULLGROUPNVAL (middle), and INFLEXION (right). The x-axis is for the presence of implied constraints and the y-axis is for the decomposition alone.

fewer failures on average. The decomposition of INFLEXION (right column) with the implied constraints is always faster than the decomposition alone, is about 35% faster on average, and has almost 100% fewer failures on average: failures are not always equal to zero, since these implied constraints do not guarantee domain consistency.

The results on the sets of unsatisfiable instances are shown in Figure 4. The decomposition of NGROUP (left column) with the implied constraints is almost always faster than the decomposition alone, is about 50% faster on average, and has about 75% fewer failures on average, even for instances of up to $m = 500$ signature variables. The decomposition of FULLGROUPNVAL (centre column) with the implied constraints is almost always faster than the decomposition alone, is about 40% faster on average, and has about 85% fewer failures on average. The decomposition of INFLEXION (right column) with the implied constraints is almost always faster than the decomposition alone, is about 60% faster on average, and has about 50% fewer failures on average. For INFLEXION, all unsatisfiable instances are detected as unsatisfiable by the decompositions with and without the implied constraints either at the root of the search tree or after failing only once, even for instances of up to $m = 200$ signature variables, that is over 10 times longer than for our experiments on satisfiable instances. For INFLEXION, about 50% of the unsatisfiable instances are not detected as unsatisfiable at the root by the decomposition alone, but are detected as unsatisfiable at the root by the decomposition with the implied constraints.

4.2 Experiments on Entire Constraint Problems

In order to test the implied constraints also in the context of entire constraint problems involving AUTOMATON constraints, we generated hard random constrained optimisation problem instances, inspired by the schemes in [22]. The generated instances have the following features:

- A set S of $s = 15$ variables, with domain $\{0, 1, 2\}$.

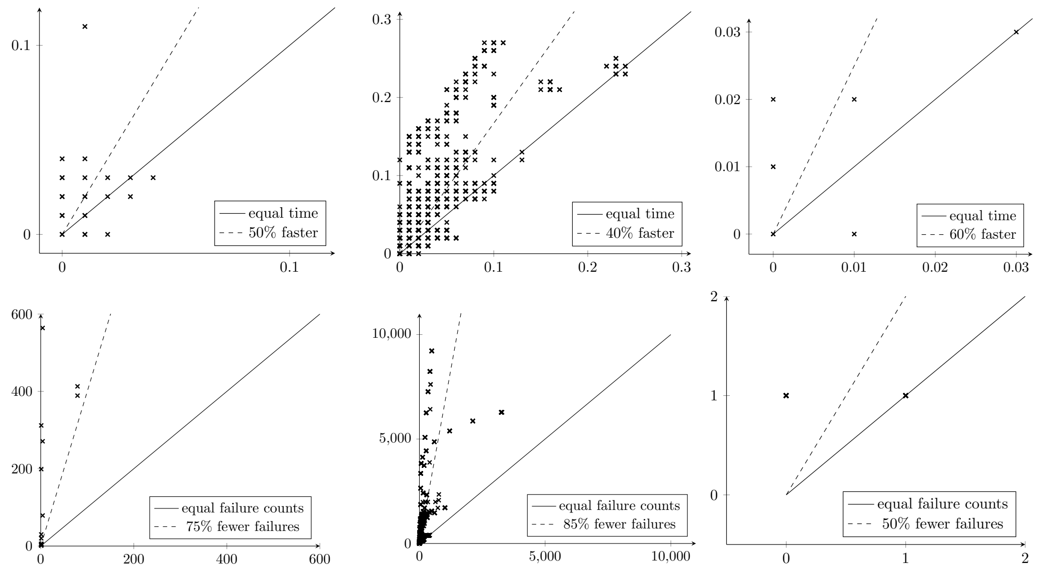


Figure 4: Seconds (top row) and failures (bottom row) to prove unsatisfiability of instances of NGROUP (left column), FULLGROUPNVAL (middle), and INFLEXION (right). The x-axis is for the presence of implied constraints and the y-axis is for the decomposition alone. Note that there really are 1,500 data points in each plot: many of them coincide.

- A sequence R of $r = 5$ variables with domain $\{0, 1, \dots, s\}$.
- A system of $\lfloor 0.5 \cdot r \cdot \ln r \rfloor = 4$ constraints, divided as follows: two constraints of the kind that is being tested (i.e., two FULLGROUPNVAL constraints, two NGROUP constraints, or two INFLEXION constraints), and two randomly picked constraints among ALLDIFFERENT, linear equalities, linear inequalities, NGROUP, FULLGROUPNVAL, and INFLEXION.
- Each NGROUP, FULLGROUPNVAL, and INFLEXION constraint is on a randomly picked subset of S and a randomly picked R_i as the result variable. An ALLDIFFERENT constraint is on a randomly picked subsequence of R. A linear equality constraint $\sum_{i=1}^{r} d_i \cdot R_i = 0$ is on R, with randomly picked coefficients $d_i \in \{-1, 0, 1\}$. A linear inequality constraint is of the form $R_i > R_j$ or $R_i > 2 \cdot R_j + R_\ell$, for randomly picked elements of R, possibly with repetition. There is no guarantee that all variables of R and S participate in some constraint. The implied constraints are added only to the decomposition of every occurrence of the constraint that is being tested.
- The cost to be maximised is the sum of the variables R_i.
- The search strategy is leftmost variable first (the R_i before the S_i), domain splitting, lower half first. Again, our purpose is orthogonal to picking a search strategy: the implied constraints do not introduce new variables and would thus not affect many search strategies, and we obtained similar results with other search strategies, as seen in Section 4.1.

We consider optimisation problems instead of finding first or all solutions to satisfaction problems because, in our particular case, when maximising the sum of the R_i, the result variables N of NGROUP and T of FULLGROUPNVAL are conflicting objectives: if N is maximal, then T is at most half the sequence length (groups of length 1); conversely, if T is maximal, then $N = 1$. However, similar results were obtained for finding all solutions to hard satisfaction problems.

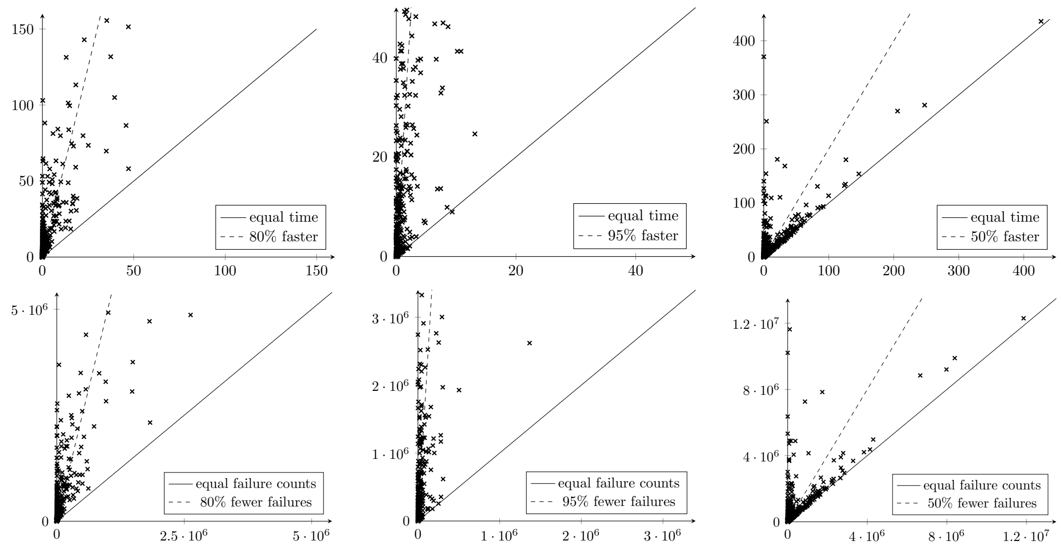


Figure 5: Seconds (top row) and failures (bottom row) to maximise a sum in problems involving nGroup (left column), FullGroupNval (middle), or Inflexion (right). The x-axis is for the decomposition with implied constraints and the y-axis is for the decomposition alone.

The results on 1,000 optimisation instances are shown in Figure 5: when the implied constraints are added to the decompositions, both time and failures are *always* reduced. This is observed even for the FullGroupNval constraint, on which the implied constraints increase the average time for satisfiable instances, as seen in Figure 3. The extra propagation provided by the implied constraints has a positive effect as the time to prove optimality is reduced, despite the potential overhead in computing the propagator fixpoint at each search tree node.

5 Conclusion, Related Work, and Future Work

We have described a fully automated parametric tool that selects, in an off-line process, a set of non-redundant linear constraints that are implied by the decomposition in [7] of a constraint on a sequence of variables, the constraint being specified by a checker provided as an automaton with linearly updated accumulators. This setting covers a large class of useful constraints. We have shown that a suitable choice, by the user, among the selected implied constraints can considerably improve solving time and propagation, both on a decomposition in isolation and on entire constraint problems containing the decomposition. With the extra propagators for the implied constraints, it potentially takes more time to compute the fixpoint of the propagators at each node of the search tree: this may backfire on the decomposition alone, but usually pays off on entire constraint problems containing the decomposition, due to the extra propagation.

The closest **related work** is [6], where we generate constraints implied by the decomposition of an Automaton($\mathcal{M}, X, R$) constraint when the variable R takes the same value whether the automaton $\mathcal{M}$ consumes the sequence X or its reverse. Like here, the implied constraints are on the accumulator variables and state variables, but they need not be linear. Unlike here, the generation is limited to the indicated particular case and is manual in most sub-cases.

Graph invariants are used in [8] to generate implied constraints automatically. In contrast, our approach does not require a database of precomputed invariants. There is also a large body of related work (e.g., [11, 12, 13, 19]) on decomposing constraints *manually* in order to maintain *domain* consistency on the decomposition. In contrast, we offer an *automatic* approach to *improving* the level of consistency on constraint decompositions.

In the **future**, we want to use a richer template than linear inequalities for implied constraints, generated in this paper by exploiting Farkas' Lemma. For instance, a non-linear template can be used by exploiting Gröbner bases. Also, disjunction enables the generation of implicative implied constraints; a motivating example is our manual derivation in [17] of such implied constraints for a constraint not discussed here, so as to achieve domain consistency actually. This will enable us to handle the full language of SICStus Prolog for accumulator updates, including the use of the 'max' and 'min' operators, thereby covering *all* the (currently 59) memory-DFAs with accumulators in the Global Constraint Catalogue.

Despite the prospects of such extensions, we have shown that even a linear template can already lead to considerable acceleration of the constraint solving process.

References

[1] E. Arafailova. Reformulation of automata for time series constraints as linear programs. Master's thesis, Mines Nantes, France, 2015.

[2] R. Barták. Modelling resource transitions in constraint-based scheduling. In W. I. Grosky and F. Plášil, editors, *SOFSEM 2002*, volume 2540 of *LNCS*, pages 186–194. Springer, 2002.

[3] N. Beldiceanu, M. Carlsson, R. Debruyne, and T. Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.

[4] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present, and future. *Constraints*, 12(1):21–62, March 2007. Catalogue at `http://sofdem.github.io/gccat`.

[5] N. Beldiceanu, M. Carlsson, R. Douence, and H. Simonis. Using finite transducers for describing and synthesising structural time-series constraints. In G. Pesant, editor, *CP 2015*, LNCS. Springer, 2015, forthcoming.

[6] N. Beldiceanu, M. Carlsson, P. Flener, M. A. Francisco Rodríguez, and J. Pearson. Linking prefixes and suffixes for constraints encoded using automata with accumulators. In B. O'Sullivan, editor, *CP 2014*, volume 8656 of *LNCS*, pages 142–157. Springer, 2014.

[7] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 107–122. Springer, 2004.

[8] N. Beldiceanu, M. Carlsson, J.-X. Rampon, and C. Truchet. Graph invariants as necessary conditions for global constraints. In P. van Beek, editor, *CP 2005*, volume 3709 of *LNCS*, pages 92–106. Springer, 2005.

[9] N. Beldiceanu, P. Flener, J. Pearson, and P. Van Hentenryck. Propagating regular counting constraints. In C. E. Brodley and P. Stone, editors, *AAAI 2014*, pages 2616–2622. AAAI Press, 2014.

[10] N. Beldiceanu, G. Ifrim, A. Lenoir, and H. Simonis. Describing and generating solutions for the EDF unit commitment problem with the ModelSeeker. In C. Schulte, editor, *CP 2013*, volume 8124 of *LNCS*, pages 733–748. Springer, 2013.

[11] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, C.-G. Quimper, and T. Walsh. Reformulating global constraints: The *slide* and *regular* constraints. In *SARA 2007*, volume 4612 of *LNAI*, pages 80–92. Springer, 2007.

[12] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decomposition of the NValue constraint. In D. Cohen, editor, *CP 2010*, volume 6308 of *LNCS*, pages 114–128. Springer, 2010.

[13] C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In C. Boutilier, editor, *IJCAI 2009*, pages 412–418. AAAI Press, 2009.

[14] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[15] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *PLILP 1997*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997. SICStus Prolog is available at http://sicstus.sics.se.

[16] M.-C. Côté, B. Gendron, and L.-M. Rousseau. Modeling the Regular constraint with integer programming. In P. Van Hentenryck and L. A. Wolsey, editors, *CP-AI-OR 2007*, volume 4510 of *LNCS*, pages 29–43. Springer, 2007.

[17] M. A. Francisco Rodríguez, P. Flener, and J. Pearson. Generation of implied constraints for automaton-induced decompositions. In A. Brodsky, E. Grégoire, and B. Mazure, editors, *ICTAI 2013*, pages 1076–1083. IEEE Computer Society, 2013.

[18] G. Pesant. A regular language membership constraint for finite sequences of variables. In M. Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

[19] C.-G. Quimper and T. Walsh. Decomposing global grammar constraints. In C. Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 590–604. Springer, 2007.

[20] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[21] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In R. Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.

[22] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8):514–534, 2007.