Implied Constraints for AUTOMATON Constraints

María Andreína Francisco Rodríguez*, Pierre Flener, and Justin Pearson

Department of Information Technology, Uppsala University, 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. An automaton with accumulators induces a decomposition of the specified constraint into a conjunction of constraints with existing propagators. Towards improving propagation, we design a fully automated tool that proposes constraints implied by such a decomposition. We show that a suitable selection of the implied constraints can considerably improve solving time and propagation.

1 Introduction

Frameworks are given in [4, 11] 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 [4]. An automaton can be seen as a *checker* for ground instances of the specified constraint.

The framework of [11] lifts an automaton into a domain-consistent *propagator* for the specified constraint. In this paper, we focus on the more general framework of [4], which lifts an automaton into a *decomposition* in terms of constraints with existing propagators. However, it is unknown how to maintain domain consistency efficiently for the decomposition. Getting closer to domain consistency is the challenge we tackle in this paper.

We continue our earlier work [10], 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 can be maintained at no asymptotic space and time overhead for the decomposition extended with implied constraints.

After a summary of the background material in Section 2, 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 [2] and proposes a set of implied constraints.

* Student supervised by the other authors.

– Our tool works *directly* on the automaton and generates implied constraints.
– Our tool eliminates uninteresting and mutually redundant constraints from the generated set of implied constraints.
– We show that a suitable set of implied constraints can improve propagation.
– Since our implied constraints are *linear*, we strongly believe they are also relevant in the context of integer-programming decompositions, such as in [9].

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

2 Background: Constraints on Automata

We define background concepts, add running examples, and state our objective.

2.1 Automata, the Regular and Automaton Constraints

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 q' upon consuming alphabet symbol σ. 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 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. A word w is *accepted* if $\widehat{\delta}(q_0, w) \in A$.

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 1. In a sequence, a *group* [2] is a maximal contiguous subsequence with values from a given set. The nGroup(X, W, N) constraint [2] holds if there are N groups of values from the given set W in the sequence X of variables. Consider the mDFA in Figure 1a. It returns the number of groups of value '$\in$' within a given word over the alphabet $\Sigma = \{\in, \notin\}$. It uses $k = 1$ accumulator. The set Q is $\{s, t\}$. The start state q_0 is s, indicated by an arrow coming from nowhere, and annotated 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 q', annotated by a symbol $\sigma \in \Sigma$ and the memory update $\langle c\rangle \coloneqq \langle c'\rangle$. If an update corresponds to the identity function, then we do not depict it. All states marked by a double circle 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. □

The Automaton$(\mathcal{M}, X, R)$ constraint [4] 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 $R = \alpha(\widehat{\delta}(\langle q_0, I\rangle, X))$.

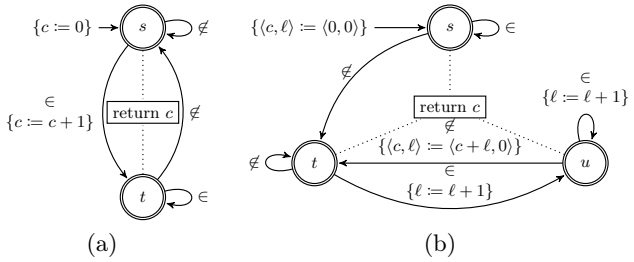

Fig. 1: (a) Memory-DFA $\mathcal{N}$ with one accumulator for $\textsc{nGroup}(X, W, N)$. (b) Memory-DFA $\mathcal{F}$ with two accumulators for $\textsc{FullGroupNval}(X, W, T)$.

2.2 Signature Variables and Signature Constraints

A constraint C on a sequence X of n variables can often be encoded with the help of a DFA or mDFA that operates not on X, but on a sequence S of m variables that are called *signature variables*, each depending via a *signature constraint* [4] under a total function on a sliding window of a consecutive variables within X. The constant $a \geq 1$ is called the *arity* of the signature constraints, and is linked to the lengths n of X and m of S by $m = n + 1 - a$.

Example 2. Consider again $\textsc{nGroup}(X, W, N)$. We constrain a sequence S of $m = n$ signature variables with domain $\{\in, \notin\}$ by the signature constraints $(X_i \in W \Leftrightarrow S_i = \text{`}\in\text{'})$ 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 Figure 1a we encode $\textsc{nGroup}(X, W, N)$ by $\textsc{Automaton}(\mathcal{N}, S, N)$ and these signature constraints. □

2.3 Decomposition of the Automaton Constraint

Consider a constraint $C(X, R)$ encoded by an $\textsc{Automaton}(\mathcal{M}, S, R)$ 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 $S_1 = X_1 \wedge \cdots \wedge S_m = X_n$, with $m = n$, to be the signature constraints. Let the mDFA $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, I, A, \alpha \rangle$ have k accumulators. The $\textsc{Automaton}(\mathcal{M}, S, R)$ constraint has the following decomposition [4]:

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

where:

- Each Q^i is a new variable, with domain Q, and is called a *state variable*: 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, and is called an *accumulator variable*: it denotes the value of the j^{th} accumulator of $\mathcal{M}$ after the values of the signature variables $S_1, \ldots, S_i$ 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 makes a transition from state q to state q', labelled by symbol $\sigma \in \Sigma$ and updating the tuple of k accumulators from the values $\langle c_1, \ldots, c_k \rangle$ to the expressions $\langle c'_1, \ldots, c'_k \rangle$; it is called a *transition constraint*.

It is unknown how to maintain domain consistency efficiently both for Trans and for this decomposition: see [1] for a detailed analysis.

2.4 Precise Formulation of Our Objective

We aim at automatically generating implied constraints that can improve propagation for the decomposition of an $\textsc{Automaton}(\mathcal{M}, S, R)$ constraint where $\mathcal{M}$ has at least one accumulator. The generation is specific to $\mathcal{M}$ but not to S and can be done off-line. We focus on mDFAs where every accumulator update is a *linear* expression on all the accumulators.. Further, we focus on generating implied constraints that are *linear* inequalities on the accumulator and state variables.

3 Generation of Linear Implied Constraints

Our approach to generating constraints implied by the decomposition of an $\textsc{Automaton}(\mathcal{M}, S, R)$ 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 L of non-linear constraints that model T being true at every state of the mDFA $\mathcal{M}$ (Section 3.1). Second, we solve L, each solution providing an instantiation of T into a particular linear implied constraint (Section 3.2). Third, we eliminate uninteresting and mutually redundant constraints from the generated set of implied constraints (Section 3.3).

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

We adapt the recipe of [12] for linear transition systems. Everything that follows requires linearity, also of the implied constraints, so we now make that restriction.

One half of Farkas' lemma (e.g., [7]) 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 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 $\beta \geq \sum_{i=1}^{e} \lambda_i b_i$. If the i^{th} linear constraint is an equality, then the requirement $\lambda_i \geq 0$ is dropped.

Let variable y_j denote the j^{th} 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 a mDFA $\mathcal{M}$ if it is true at the start state and if its truth is preserved by every transition of $\mathcal{M}$. For the start state, we encode using Farkas' lemma that the point-wise initialisation equalities

behind $\langle y_1, \ldots, y_k \rangle = I$ have T as a logical consequence. For each transition $\delta(\langle q, \langle y_1, \ldots, y_k \rangle \rangle, \sigma) = \langle q', \langle y'_1, \ldots, y'_k \rangle \rangle$, where each y'_j is a linear functional expression, 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.

We now go beyond adapting the recipe of [12], by discussing three refinements of the ideas seen so far. First, many implied constraints that provide extra propagation are expressed on the *current* and *previous* values of the accumulators.

Example 3. 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 [10], 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 $\alpha_1 = -1 \wedge \alpha_2 = 0 \wedge \alpha_3 = 1 = \beta$. □

Our tool allows the user to indicate the history length.

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. Third, 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. Our tool allows the user to switch on these options.

3.2 Implied Constraints: Generation by Solving the System L

We now show how to solve L 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 it suffices to solve the resulting non-linear constraint system for *integer* values of the variables. Further, we reckon that for each variable a small finite integer interval centred on zero, such as $\{-5, \ldots, 5\}$, suffices for finding many useful implied constraints. Hence we solve the constraints using a *finite*-domain CP solver. Since our tool is written in SICStus Prolog and reads automata in the SICStus Prolog syntax, we use SICStus Prolog [8].

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

Example 4. Consider the implied constraint $c \leq c_2 + 1$ of Example 3. 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 $c - c_1 \leq q$. It turns out that $c - c_1 \leq q$ is an implied constraint, requiring c and c_1 to be equal at the start state s and apart by at most one unit at state t. 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 L_2. It turns out that $c \leq c_2 + 1$ is now an implied constraint, generated from a solution to L_2. □

Our tool allows the user to indicate an upper bound 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.

3.3 Implied Constraints: Redundancy Elimination and Proposal

Some generated implied constraints are useless. For example, the implied constraint $5 \geq 0$ is vacuously true. Other generated implied constraints are mutually redundant. For example, the implied constraint $c \leq c_2 + 1$ is redundant with $3c \leq 3c_2 + 3$. Our tool automatically eliminates useless and redundant constraints.

Since the decomposition (1) of $\textsc{Automaton}(\mathcal{M}, [S_1, \ldots, S_m], R)$ reveals accumulator variables C_j^i and state variables Q^i for every prefix $[S_1, \ldots, S_i]$, with $0 \leq i \leq m$, we advocate posting an implied constraint 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 the decompositions in isolation (Section 4.1) and in the context of entire constrained optimisation problems (Section 4.2). These experiments were run using the following implied constraints: $c \geq c_2$ and $c_2 + 1 \geq c$ for nGroup, $c \geq c_1$ and $c_4 + \ell_4 + 4 \geq c + \ell$ for FullGroupNval [2] (using the mDFA of Figure 1b), as well as $c_1 + 1 \geq c$ and $c_2 + 2 \geq c$ for Inflexion [6].

All experiments were run in SICStus Prolog 4.2 [8] 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 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 $a = 3$). The instances for nGroup have sequences of length $m = 100$, those for FullGroupNval have $m = 50$, and those for Inflexion have $m = 15$. We generated a set of 1,500 satisfiable instances and a set of 1,500 unsatisfiable instances. The default search strategy is used: leftmost variable first (R before S), lower value first.

The results on the sets of satisfiable instances are shown in Figure 2. The decomposition of nGroup with the implied constraints is almost always faster than the decomposition alone, is about 30% faster, and has about 60% fewer failures on average. The decomposition of FullGroupNval with the implied constraints

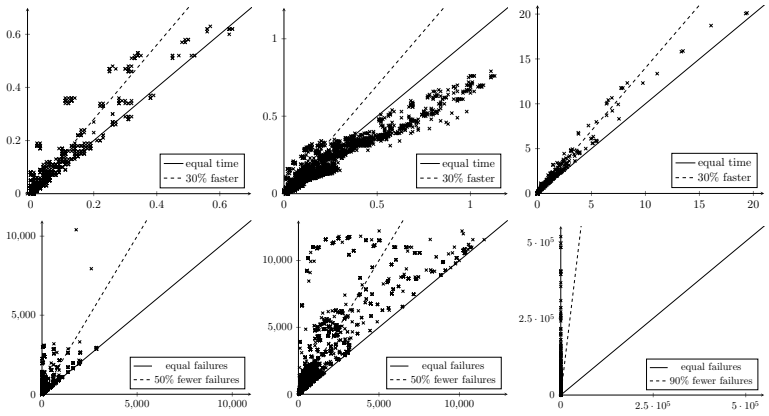

Fig. 2: Seconds (top row) and failures (bottom row) to find all solutions to satisfiable instances of nGroup (left column), FullGroupNval (centre), and Inflexion (right). The x-axis is for the presence of implied constraints.

is sometimes faster than the decomposition alone, but is about 20% slower, and has about 50% fewer failures on average. The decomposition of Inflexion with the implied constraints is always faster than the decomposition alone, is about 40% faster, and has almost 100% fewer failures on average.

The results on the sets of unsatisfiable instances are similar to those on the sets of satisfiable instances, and have been omitted for space reasons.

4.2 Experiments on Entire Constraint Problems

In order to test the implied constraints also in the context of entire constraint problems, we generated hard random constrained optimisation problem instances, inspired by the schemes in [13]. The instances have the following features:

- A set S of $s = 15$ variables, with domain $\{0, 1, 2\}$.
- A sequence R of $r = 5$ variables with domain $\{0, 1, \ldots, s\}$.
- A system of $\lfloor 0.5 \cdot r \cdot \ln r \rfloor = 4$ constraints, divided as follows:
  - 2 constraints of the kind that is being tested (i.e., 2 FullGroupNval constraints, 2 nGroup constraints, or 2 Inflexion constraints),
  - 2 randomly selected constraints among AllDifferent, linear equalities, linear inequalities, nGroup, FullGroupNval, and Inflexion.
- Each nGroup, FullGroupNval, and Inflexion constraint is on a randomly selected subset of S and a randomly selected R_i as the result variable. An AllDifferent constraint is on a randomly selected subsequence of R. A linear equality constraint $\sum_{i=1}^{r} d_i \cdot R_i = 0$ is on R, with randomly selected 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 selected elements of R, possibly with repetition. The implied constraints are added only to the decomposition of every occurrence of the constraint that is being tested.

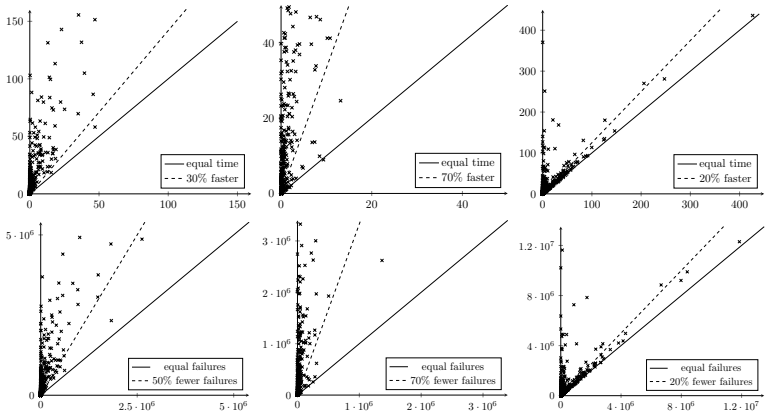

Fig. 3: Seconds (top row) and failures (bottom row) to maximise a sum in 1,000 problem instances involving nGroup, FullGroupNval, or Inflexion.

- The cost to be maximised is the sum of the variables R_i.
- The search strategy is leftmost variable first (R before S), domain splitting, lower half first.

The results are shown in Figure 3. When the implied constraints are added to the decompositions, both the time and the failures are *always* reduced.

5 Conclusion, Related Work, and Future Work

We have described a fully automated parametric tool that proposes, in an off-line process, a set of non-redundant linear constraints that are implied by the decomposition in [4] of a constraint being specified by an automaton with linearly updated accumulators. We have shown that a suitable selection of the proposed implied constraints can considerably improve solving time and propagation.

The closest **related work** is [3], 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 [5] to generate implied constraints automatically. In contrast, our approach does not require a database of precomputed invariants.

In the **future**, we want to use a richer template for implied constraints.

References

1. Beldiceanu, N., Carlsson, M., Debruyne, R., Petit, T.: Reformulation of global constraints based on constraints checkers. Constraints 10(4), 339–362 (2005)
2. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present, and future. Constraints 12(1), 21–62 (March 2007), the catalogue is at `http://sofdem.github.io/gccat`
3. Beldiceanu, N., Carlsson, M., Flener, P., Francisco Rodríguez, M.A., Pearson, J.: Linking prefixes and suffixes for constraints encoded using automata with accumulators. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 142–157. Springer (2014)
4. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer (2004)
5. Beldiceanu, N., Carlsson, M., Rampon, J.X., Truchet, C.: Graph invariants as necessary conditions for global constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 92–106. Springer (2005)
6. Beldiceanu, N., Ifrim, G., Lenoir, A., Simonis, H.: Describing and generating solutions for the EDF unit commitment problem with the ModelSeeker. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 733–748. Springer (2013)
7. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2004)
8. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer (1997), SICStus Prolog is available at `http://sicstus.sics.se`
9. Côté, M.C., Gendron, B., Rousseau, L.M.: Modeling the Regular constraint with integer programming. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CP-AI-OR 2007. LNCS, vol. 4510, pp. 29–43. Springer (2007)
10. Francisco Rodríguez, M.A., Flener, P., Pearson, J.: Generation of implied constraints for automaton-induced decompositions. In: Brodsky, A., Grégoire, E., Mazure, B. (eds.) ICTAI 2013. pp. 1076–1083. IEEE Computer Society (2013)
11. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer (2004)
12. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004, LNCS, vol. 3148, pp. 53–68. Springer (2004)
13. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence 171(8), 514–534 (2007)