Fundamenta Informaticae 107 (2011) 229–254
DOI 10.3233/FI-2011-401
IOS Press

An *automaton* Constraint for Local Search *

Jun He†, **Pierre Flener**, **Justin Pearson**
Department of Information Technology
Uppsala University, Box 337, SE-751 05 Uppsala, Sweden
{*Jun.He; Pierre.Flener; Justin.Pearson*}*@it.uu.se*

Abstract. We explore the idea of using automata to implement new constraints for local search. This is already a successful approach in constraint-based global search. We show how to maintain the violations of a constraint and its variables via a deterministic finite automaton that describes a ground checker for that constraint. We extend the approach to counter automata, which are often much more convenient than finite automata, if not more independent of the constraint instance. We establish the practicality of our approach on several real-life combinatorial problems.

Keywords: Automaton constraint · Counter automaton · Local search

1. Introduction

When a high-level constraint programming (CP) language lacks a (possibly global) constraint that would allow the formulation of a particular model of a combinatorial problem, then the modeller traditionally has the choice of (1) switching to another CP language that has all the required constraints, (2) formulating a different model that does not require the lacking constraint, or (3) implementing the lacking constraint in the low-level implementation language of the chosen CP language. This paper addresses the core question of facilitating the third option, and as a side effect often makes the first two options unnecessary.

The user-level extensibility of CP languages has been an important goal for over a decade. In the traditional global search approach to CP (namely heuristic-based tree search interleaved with propagation), higher-level abstractions for describing new constraints include indexicals [16]; (possibly enriched)

*This paper is an extension of [6], which was also presented at RCRA'09.
†Address for correspondence: Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden

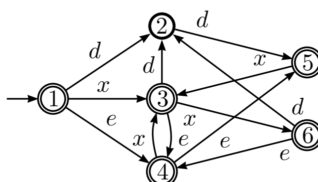

Figure 1. An automaton for a simple work scheduling constraint.

deterministic finite automata (DFAs) via the $automaton$ [2] and $regular$ [11] generic constraints; and multi-valued decision diagrams (MDDs) via the mdd [4] generic constraint. Usually, a general but efficient propagation algorithm achieves a suitable level of local consistency by processing the higher-level description of the new constraint. In the more recent local search approach to CP (called constraint-based local search, CBLS, in [13]), higher-level abstractions for describing new constraints include invariants [10]; a subset of first-order logic with arithmetic via combinators [15] and differentiable invariants [14]; and existential monadic second-order logic for constraints on set variables [1]. Usually, a general but incremental algorithm maintains the constraint and variable violations by processing the higher-level description of the new constraint.

Example 1.1. In Figure 1 we give our running example. It is a deterministic finite automaton (DFA; see [7], for example) that describes a simple work scheduling constraint. There are values for two work shifts, namely day (d) and evening (e), as well as a value for enjoying a day off (x). Work shifts are subject to the following four conditions: one must take at least one day off before a change of work shift; if one works on a day shift, then one must do so for exactly two consecutive days; one can work on at most two consecutive evening shifts; and one cannot enjoy more than two consecutive days off. The initial state 1 is marked by a transition entering from nowhere, while the final states 1, 3, 4, 5, and 6 are marked by double circles. Missing transitions, say from state 2 upon reading value e, are assumed to go to an implicit failure state, with a self-looping transition for every value (so that no final state is reachable from it). The set of strings accepted by the automaton defines the set of acceptable work shift sequences.

In this paper, we revisit the description of new constraints via automata, already successfully tried within the global search approach to CP [2, 11], and show that it can also be successfully used within the local search approach to CP, as also argued by [12]. The significance of this endeavour can be assessed by noting that 119 of the currently 348 global constraints in the *Global Constraint Catalogue* [3] are described by DFAs that are possibly enriched with counters [2], so that all these constraints are instantly made available in CBLS.

The $automaton(X, A)$ generic constraint of this paper is satisfied when automaton A accepts the sequence X of variables (unknowns). The contributions and organisation of the remainder of this paper are as follows:

- We present two new algorithms for maintaining the violation of the $automaton(X, A)$ constraint (the automaton A is deterministic) and the violations of the variables of that constraint (Section 2).
- We extend the approach to counter automata, which save modelling time by being more compact if not more generic, that is more independent of the instance of the described constraint. Our counter automata generalise those in [2] (Section 3).

- Our algorithm for unwinding a counter automaton (when it recognises a regular language) into a counter-free automaton generalises many existing ad hoc constructions of automata from instances of a constraint (Section 3.2).
- We present experimental results establishing the practicality of our results, also in comparison to the prior CBLS results of [12] and to handcrafted constraints (Section 4).
- We propose violation algorithms for the $stretch_path$ and $stretch_path_partition$ global constraints (Section 4.1).

Finally, in Section 5, we summarise this work and discuss related as well as future work.

2. Violation Maintenance with Counter-Free Automata

In CBLS [10, 13], constraints are used to describe and control local search. Given an initial assignment of values to all the variables, CBLS tries to find a better assignment that decreases the amount of constraint violation, by searching a neighbourhood of the current assignment, that is a set of assignments that do not differ much from the current one. A solution with zero (or minimal) violation is to be found. Meta-heuristics are used to escape local minima. Search heuristics can be guided by two related measures of violation: for each constraint, a measure of *constraint violation* and a measure of *variable violation*. Constraint violation measures how close the constraint is to being satisfied. Variable violation measures for each variable in the constraint the variation of the constraint violation that could be achieved if that variable was suitably modified. Although these terms are not formally defined here, it is possible for a large number of constraints [13] to come up with heuristically useful definitions of constraint and variable violations. Three things are required to implement a constraint: a method for calculating the violation of the constraint and each of its variables for the initial assignment; a method for computing the differences of these violations upon a candidate move to a neighbouring assignment; and a method for incrementally maintaining these violations when an actual move is made. Since in local search the constraint and variable violations might need to be calculated thousands of times so as to pick the best move, the algorithms implementing these must be very efficient and, where possible, incremental.

2.1. Violations of a Constraint

To define and compute the violations of a constraint described by an automaton, we first introduce the notion of a segmentation of an assignment:

Definition 2.1. (Segmentation) Given a sequence $X = \langle X_1, \ldots, X_n \rangle$ of n variables assigned to the sequence $\langle d_1, \ldots, d_n \rangle$ of values, a *segmentation* is a possibly empty sequence of non-empty sub-strings (referred to here as *segments*) $\sigma_1, \ldots, \sigma_\ell$ of the string $d_1 \cdots d_n$ such that for all $k > j$ and segments $\sigma_j = d_p \cdots d_q$ and $\sigma_k = d_r \cdots d_s$ we have that $r > q$.

For example, given the sequence $\langle x, x, d, e, x, x \rangle$ of values, a possible segmentation is $\langle x, x, d \rangle, \langle x, x \rangle$; note that, in this segmentation, the fourth character of the assignment is not part of any segment. In general, an assignment has multiple possible segmentations. We are interested in segmentations that are accepted by an automaton, in the following sense:

Definition 2.2. (Acceptance) Given an automaton and a sequence $\langle d_1, \ldots, d_n \rangle$ of values, a segmentation $\sigma_1, \ldots, \sigma_\ell$ is *accepted* by the automaton if there exist strings $\alpha_1, \ldots, \alpha_{\ell+1}$, where only α_1 and $\alpha_{\ell+1}$ may be empty, satisfying the following two conditions. First, the concatenated string

$$\alpha_1 \cdot \sigma_1 \cdot \alpha_2 \cdot \ \cdots \ \cdot \alpha_\ell \cdot \sigma_\ell \cdot \alpha_{\ell+1} = e_1 \cdots e_n$$

is accepted by the automaton. Second, for all segments $\sigma_j = d_p \cdots d_q$ and for all $p \leq k \leq q$, we have that $e_k = d_k$. That is, the strings $\alpha_1, \ldots, \alpha_{\ell+1}$ are of the correct length so that the new concatenated string $e_1 \cdots e_n$ has the segments $\sigma_1, \ldots, \sigma_\ell$ in the same place as in $d_1 \cdots d_n$.

For example, given the automaton in Figure 1, the sequence $\langle x, x, d, e, x, x \rangle$ of values has a segmentation $\langle x, x, d \rangle, \langle x, x \rangle$ with $\ell = 2$ segments, which is accepted by the automaton via the string $\langle x, x, d, d, d, x, x \rangle$ with $\alpha_1 = \alpha_3 = \epsilon$ (the empty string) and $\alpha_2 = \langle d \rangle$.

Given an assignment, the constraint and variable violations are defined relative to a given segmentation.

Definition 2.3. (Violations) Given an automaton describing a constraint c and given a segmentation $\sigma_1, \ldots, \sigma_\ell$ of a sequence $\langle d_1, \ldots, d_n \rangle$ of values for a sequence of n variables $\langle X_1, \ldots, X_n \rangle$:

- The *constraint violation* of c is $n - \sum_{j=1}^{\ell} |\sigma_j|$ (where $|\sigma|$ denotes the length of segment σ).
- The *variable violation* of variable X_i is 0 if there exists a segment $\sigma_j = d_p \cdots d_q$ such that $p \leq i \leq q$, and 1 otherwise, in which case we say that X_i is *violated*.

Proposition 2.1. If the violation of a constraint with respect to a given segmentation is zero, then the current assignment is a satisfying assignment.

Proof: Let the violation of a constraint be zero, and let $\alpha_1 \cdot \sigma_1 \cdot \alpha_2 \cdot \ \cdots \ \cdot \alpha_\ell \cdot \sigma_\ell \cdot \alpha_{\ell+1}$ be an accepted string. According to Definition 2.3, $n - \sum_{j=1}^{\ell} |\sigma_j|$ is zero, so $\sigma_1 \cdot \sigma_2 \cdot \ \cdots \ \cdot \sigma_\ell$ is of length n and is an accepted string. Based on Definition 2.2, the current assignment is a satisfying assignment. □

In practice, in order to have a termination criterion, we want the violation of a constraint to *decide* whether the current assignment is satisfying or not, that is we also want the converse of Proposition 2.1. In Section 2.2, we show that this is the case for the segmentations computed by our approach.

Proposition 2.2. The violation of a constraint with respect to a given segmentation is the sum of the violations of its variables.

Proof: Observe that for an accepted string $\alpha_1 \cdot \sigma_1 \cdot \alpha_2 \cdot \ \cdots \ \cdot \alpha_\ell \cdot \sigma_\ell \cdot \alpha_{\ell+1}$, as in Definition 2.2, each violated variable only corresponds to some α_i. The result then follows from the fact that

$$n = \sum_{i=1}^{\ell+1} |\alpha_i| + \sum_{i=1}^{\ell} |\sigma_i|$$

□

The Hamming distance between the current assignment and a satisfying assignment (an assignment whose string is accepted by the automaton) is the number of variables whose values have to change in order to satisfy the constraint described by the automaton. We have the following result:

Proposition 2.3. The violation of a constraint with respect to a given segmentation is at least the minimal Hamming distance between the current assignment and any satisfying assignment.

Proof: Given a segmentation $\sigma_1, \ldots, \sigma_\ell$ of the current assignment, there exists an accepted string $\alpha_1 \cdot \sigma_1 \cdot \alpha_2 \cdot \cdots \cdot \alpha_\ell \cdot \sigma_\ell \cdot \alpha_{\ell+1}$, as in Definition 2.2. The constraint violation with respect to the given segmentation is $n - \sum_{j=1}^{\ell} |\sigma_j|$, which is the Hamming distance between the current assignment and the accepted string. Then, the constraint violation can never be smaller than the minimal Hamming distance to any accepted string (satisfying assignment). □

In other words, a segmentation never underestimates the number of variables that have to change to reach a solution, and usually overestimates. A more precise measure often leads to fewer iterations to find a solution; however if the computation of such a measure is costly, there may be overall a loss of time. As shown in our experimental results in Section 4, our simple measure works much better than a precise but computationally expensive measure.

Our approach, described in the next two sub-sections, greedily grows a segmentation from left to right relative to a satisfying assignment, and makes stochastic choices whenever greedy growth is impossible.

2.2. Calculating the Violations

Our algorithms calculate the constraint and variable violations stochastically, in time linear in the number n of variables. The first version of the algorithm unrolls the automaton into a layered graph, specific to n, in order to ease computation, as in [11].

Definition 2.4. (Layered Graph) Given a finite automaton with m states, the *layered graph* over a given number n of variables is a graph with $m \cdot (n+1)$ nodes. Each of the $n+1$ vertical layers has a node for each of the m states of the automaton. The node for the initial state of the automaton in layer 1 is called the start node. There is an arc labelled w from node f in layer i to node t in layer $i+1$ if and only if there is a transition labelled w from f to t in the automaton. A node in layer $n+1$ is marked as a success node if it corresponds to a final state in the automaton.

The layered graph is further processed by removing all nodes and arcs that do not lead to a success node. The resulting graph, seen as a DFA (or as an ordered MDD), need *not* be minimised (or reduced) for our approach (although this is a good idea for the global search approaches [2, 11], as argued in [8], and would be a good idea for the local search approach of [12]), as the number of arcs of the graph does not influence the time complexity of our algorithms below.

For instance, the unrolled version for $n = 6$ variables of the automaton in Figure 1 is given in Figure 2. Note that a satisfying assignment that the variables are assigned to the sequence $\langle d_1, \ldots, d_n \rangle$ of values corresponds to a path from the start node in layer 1 to a success node in layer $n+1$, such that each arc from layer i to layer $i+1$ of this path is labelled d_i.

The algorithms to calculate the violations require a number of data structures. Throughout this section, let m denote the number of states in the given automaton and let n denote the number of variables:

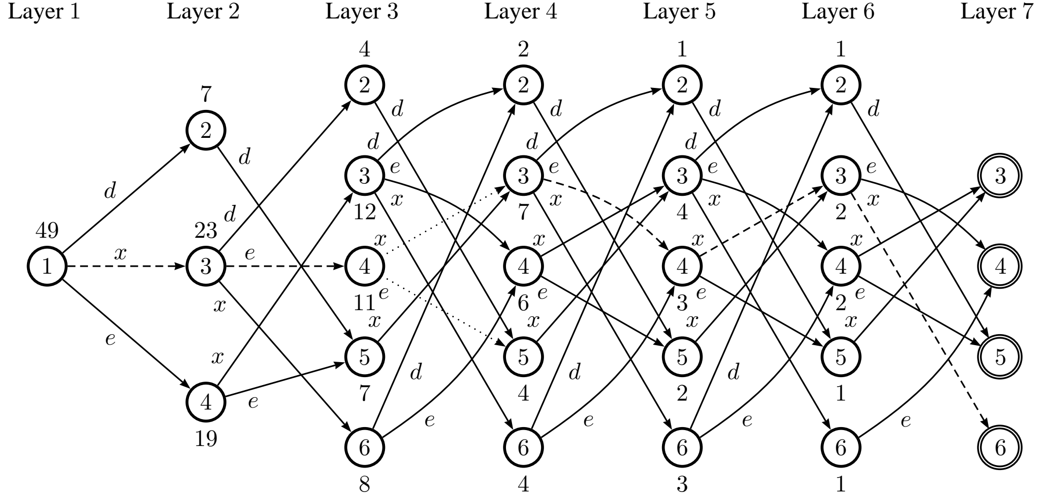

Figure 2. The unrolled automaton of Figure 1. The number by each node is the number of paths from that node to a success node in the last layer.

- $nbrPaths[1 \leq i \leq n, 1 \leq j \leq m]$ records the number of paths from node j in layer i to a success node in the last layer, counted in the same way as in [20]; for example, see the numbers by each node in Figure 2;
- ℓ is the number of segments in the current segmentation;
- $\sigma_1, \ldots, \sigma_\ell$ are the segments of the current segmentation;
- $Violation[1 \leq i \leq n]$ records the current violation of variable X_i;

The $nbrPaths$ matrix can be computed straightforwardly by dynamic programming. The other three data structures are initialised (when the starting position is $s = 1$) and maintained (when variable X_s is changed, with $s \geq 1$) by the $calcSegment(s)$ procedure of Algorithm 1. Via some initialisations (lines 2 and 3), it (re)visits only the variables $X_s, \ldots, X_n$ (line 4). If the value of the currently visited variable X_i triggers the extension of the currently last segment or the creation of a new segment (lines 6 to 8), then its violation is 0 (line 9). Otherwise, its violation is 1 and a successor node is picked with a probability weighted according to the number of paths from the current node to a success node (lines 12 to 13). Toward this, we maintain the nodes of the picked path (line 14). Picking a successor deterministically (say always the first successor) is much less efficient and requires the search space to be highly connected.

The time complexity of Algorithm 1 is linear in the number n of variables, because only one path (from layer s to layer $n+1$) is explored, with a constant-time effort at each node. Once the pre-processing is done, the time complexity of Algorithm 1 is *dependent* on the depth of the unrolled automaton and is *independent* of the number of arcs of the unrolled automaton. Hence the minimisation (or reduction) of the unrolled automaton would be merely for space savings (and for the convenience of human reading) as well as for accelerating the pre-processing computation of the $nbrPaths$ matrix. In our experiments (not reported here for space reasons), these space and time savings are not warranted by the amount of time

Algorithm 1 Initialisation and update of the segmentation from position s

1: **procedure** $calcSegment(s)$
2: let ℓ be the number of segments picked for $\langle V_1, \ldots, V_{s-1}\rangle$ at the previous run; assume $\ell = 0$ at the first run
3: $node[1] \leftarrow 1$; $inSegment \leftarrow \mathbf{true}$
4: **for all** $i \leftarrow s$ **to** n **do**
5:   **if** the current value, say a, of V_i is the label of an arc from $node[i]$ to t **then**
6:     **if not** $inSegment$ **then**
7:       $\ell \leftarrow \ell + 1$; $\sigma_\ell \leftarrow \epsilon$; $inSegment \leftarrow \mathbf{true}$ {create a new segment}
8:     $\sigma_\ell \leftarrow \sigma_\ell \cdot a$
9:     $Violation[i] \leftarrow 0$
10:   **else**
11:     $inSegment \leftarrow \mathbf{false}$
12:     $Violation[i] \leftarrow 1$
13:     pick a successor t of $node[i]$ with probability $\frac{nbrPaths[i+1,t]}{nbrPaths[i,node[i]]}$
14:   $node[i+1] \leftarrow t$

required for minimisation (or reduction), and minimisation here never reduces the depth of the unrolled automaton.

Proposition 2.4. If the current assignment is a satisfying assignment, then the constraint violation with respect to the segmentation computed by Algorithm 1 is zero.

Proof: Let a sequence $\langle X_1, \ldots, X_n\rangle$ of n variables be assigned to a sequence $\langle d_1, \ldots, d_n\rangle$ of values, and let the string $d_1 \cdots d_n$ be accepted by the automaton. According to Definition 2.4, there exists a path $node[1] \rightarrow \cdots \rightarrow node[n+1]$ in the layered graph, where $node[1]$ is the start node and $node[n+1]$ is a success node, such that there exists an arc labelled with value d_i between $node[i]$ and $node[i+1]$ for any $1 \leq i \leq n$. The condition on line 5 of Algorithm 1 is always satisfied under the current assignment, and a segmentation with one segment $\sigma_1 = \langle d_1, \ldots, d_n\rangle$ is computed. The constraint violation is zero with respect to the segmentation. □

Example 2.1. In Figure 2, with the initial assignment that all variables are assigned to the sequence $\langle x, e, d, e, x, x\rangle$ of values and a first call to Algorithm 1 with $s = 1$, the first segment is $\langle x, e\rangle$ (the first dashed path). Next, the assignment $X_3 := d$ triggers a violation of 1 for variable X_3 because there is no arc labelled d that connects the current node 4 in layer 3 with any nodes in layer 4. However, node 4 in layer 3 has two out-going (dotted) arcs, namely to nodes 3 and 5 in layer 4. In layer 4, there are 7 paths from node 3 to the last layer, compared to 4 such paths from node 5, so node 3 is picked with probability $\frac{7}{11}$ and node 5 is picked with probability $\frac{4}{11}$ (where the 4, 7, and 11 are the numbers by those nodes), and we assume that node 3 in layer 4 is picked. From there, we get the second segment $\langle e, x, x\rangle$ (the second dashed path), which stops at success node 6 in the last layer. The violation of the constraint is thus 1, because the value of one variable does not participate in any segment.

Assume now that variable X_3 is changed to value e, and hence we call Algorithm 1 with $s = 3$. Only $\ell = 1$ segment can be kept from the previous segmentation picked for $\langle X_1, X_2\rangle$, namely $\langle x, e\rangle$ (the first

dashed path). Since there is an arc labelled e from the current node 4 in layer 3, namely to node 5 in layer 4, segment σ_1 is extended (line 8) to $\langle x, e, e \rangle$. However, with variable X_4 still having value e, this segment cannot be extended further, since there is no arc labelled e from node 5 in layer 4, and hence X_4 is violated. The sole successor node 3 in layer 5 is chosen; as there is an arc labelled x from the current node to node 6 in layer 6, a new segment $\sigma_2 = \langle x \rangle$ is created. However, variable X_6 is violated, and segment σ_2 cannot be extended further. Because the values of two variables do not participate in any segment, the violation of the constraint is 2. It is larger than the minimal Hamming distance between the current assignment and any satisfying assignment, as there exists a satisfying assignment, that the variables are assigned to $\langle x, e, x, e, x, x \rangle$, and the Hamming distance between $\langle x, e, e, e, x, x \rangle$ and $\langle x, e, x, e, x, x \rangle$ is 1. Hence changing variable X_3 from value d to value e would not be considered a good move, as the constraint violation increases from 1 to 2. Changing variable X_3 to value x instead would be a much better move, as the first segment $\langle x, e \rangle$ is then extended to the entire current assignment, that the variables are assigned to $\langle x, e, x, e, x, x \rangle$, without detecting any violated variables, so that the violation of the constraint is then 0, meaning that a satisfying assignment was found.

2.3. Depth-First Search

Algorithm 1 requires that the automaton be unrolled, and further the unrolled graph has to have all paths removed that do not lead to a success node. The size of the unrolled automaton is proportional to the product of the number of variables and the number of states of the automaton. If there is a large number of variables, then the unrolled automaton could be very large. Using depth-first search (DFS), the graph can be dynamically unrolled.

This can be done by simply modifying Algorithm 1 so that whenever a successor node is checked, we perform a DFS to check for each outgoing arc if there is a path of the correct length to a success node. The extra worst-case cost of DFS can be amortised by caching (often referred to as memorisation) the results of previous DFSs. A further simplification is made to line 13, which picks a successor using a weighted random choice based on the number of paths to a success node: as the layered graph contains nodes and arcs that do not lead to a success node, we cannot cheaply calculate dynamically the number of paths to a success node as in Section 2.2, and we here simply pick a random successor node among all the successor nodes that have been shown via DFS to have a path to a success node.

2.4. Related Work

The only related work we are aware of is a CBLS implementation [12] in COMET [13] of the $regular$ constraint [11], based on the ideas for the (global search) propagator of the $soft_regular$ constraint [17]. The difference is that, upon a candidate move, they estimate the violation change compared to the *nearest* satisfying assignment (in terms of Hamming distance from the current assignment), whereas we estimate it compared to *a* randomly picked satisfying assignment. In our terminology (although it is not implemented that way in [12]), they find a segmentation such that an accepted string for the automaton has the minimal Hamming distance to the current assignment.

It is always possible to state a new global constraint using the differentiable invariants [14] of COMET: it suffices to encode all the paths from the start node to a success node of the (ideally minimised) unrolled automaton for that constraint by using COMET's conjunction and disjunction combinators. However, as the automaton or the number of variables to unroll for gets larger, this expression can

become too large to post, and even when it can be posted, our experiments (not reported here for space reasons) show that our approach is much more efficient for large enough unrolled automata.

3. Violation Maintenance with Counter Automata

In Section 3.1, we propose the concept of counter automaton (cDFA) as a more convenient and generic way of describing a new global constraint than a deterministic finite automaton (DFA). We show that cDFAs accept also non-regular languages. In practice, we are here only interested in finite languages (of words of a given length), so we will not exploit this additional expressiveness. There are two ways of using counter automata in local search. In Section 3.2, we show how to unwind a cDFA (accepting a regular language) into a DFA in an off-line pre-processing step, so that the methods in Section 2 can be re-used and that the cDFA itself is purely for the convenience of modelling. In Section 3.3, we generalise the DFS-based violation algorithm for DFAs of Section 2.3 to work directly on cDFAs.

3.1. Counter Automata

A counter automaton (cDFA) is defined just like a deterministic finite automaton (DFA), except that the transitions can include assignment statements to counter variables and that the transitions and final states can be guarded by conditions on these counters. The "transition" to the initial state has unguarded initialisations of the counters. Given the transition function δ, a guarded transition $\delta(q, a, \alpha, \beta) = t$, whose graphical representation is an arc annotated with "$a\ \{\alpha \to \beta\}$" from state q to state t, means that if symbol a is read and guard α holds at state q, then state t is reached, upon also executing the counter assignment statements β. In principle, a guard α can be any decidable logical expression of comparison and membership atoms among counters and parameters of the constraint. Similarly, a counter assignment statement β can be any computable sequential or conditional composition of arithmetic operations on the counters, or a no-operation (denoted nop). The guards of transitions on the same symbol from the same state must be mutually exclusive to make the counter automaton deterministic. A guarded final state (q, α), whose graphical representation is "$q : \alpha$" within a double ellipsis, means that state q is final only if guard α holds. As usual, we sometimes abbreviate the graphical representation of several arcs between the same pair of states by a single arc annotated with the set of symbols of those arcs, provided they have the same guards and counter statements.

Counter automata provide a powerful tool for modelling. Indeed, a DFA is often specific to an instance of that constraint, as seen in the following example:

Example 3.1. Reconsider the work scheduling constraint in Example 1.1: if we change just one parameter of that constraint, say that one cannot work for fewer than two or more than four consecutive days on the day shift, instead of exactly two days, then the DFA in Figure 1 needs to be changed. Although the only difference of these two instances of the work constraint is just a single parameter, their DFAs have many differences. It will require a lot of modelling work if every constraint instance needs a different automaton. However, cDFAs are often independent of constraint instances. Figure 3 gives a cDFA for all instances of the work constraint requiring that one cannot work for fewer than $\underline{d}$ or more than $\overline{d}$ consecutive days on the day shift; that one cannot work for fewer than $\underline{e}$ or more than $\overline{e}$ consecutive days on the evening shift; and that between any change of work shifts, one must enjoy at least $\underline{x}$ and at most $\overline{x}$ consecutive days off. The counter c maintains the number of consecutive days on the same shift.

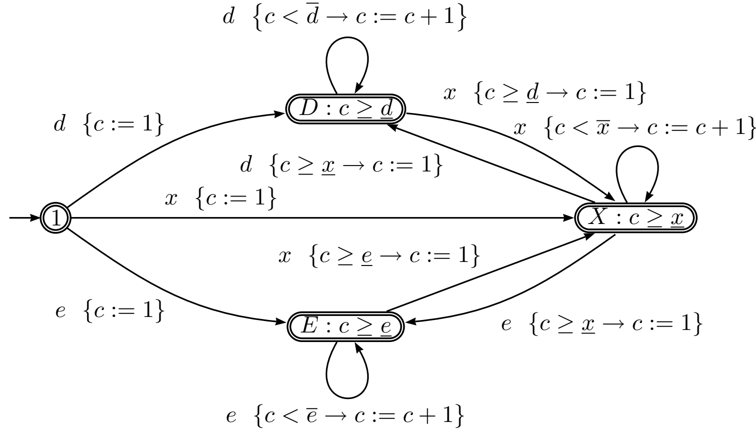

Figure 3. A counter automaton for the work scheduling constraint in Figure 1.

States D, E, and X are guarded final states: for example, D is a final state only if $c \geq \underline{d}$. The guarded transition $d\ \{c < \overline{d} \to c := c+1\}$ from D to itself means that the transition on symbol d fires if $c < \overline{d}$ and increments c by one. The unguarded transition $d\ \{c := 1\}$ from state 1 to D means that the transition on d always fires and initialises c to 1. In Section 3.2, we give a general algorithm that can automatically unwind this cDFA for parameters $\langle \underline{d}, \overline{d}, \underline{e}, \overline{e}, \underline{x}, \overline{x} \rangle = \langle 2, 2, 1, 2, 1, 2 \rangle$ into the DFA of Figure 1.

Many more cDFAs, and their unwound DFAs, are given in the experiments of Section 4, which thus also aims at showing the great modelling convenience of counter automata.

Every DFA is also a cDFA, namely a cDFA without counters and guards. Many classical algorithms for DFAs, such as product (see [7], for example), straightforwardly generalise to cDFAs, as the guards and counter assignments can be just carried along as annotations. Counter automata are more expressive than DFAs, which can only recognise regular languages. Indeed, it is possible to design cDFAs to recognise the language $a^n b^n$, which is context-free but not regular, and the language $a^n b^n c^n$, which is not even context-free. Note that our abbreviation "cDFA" thus just indicates that a DFA was annotated by counters, but not that only regular languages are recognised.

Our counter automata are more general than those of the *Global Constraint Catalogue* [2, 3], where it is the counter assignments that are guarded (rather than the transitions) and where the transition on a given symbol from a given state is unique and fires unconditionally.

3.2. Unwinding a Counter Automaton

We propose Algorithm 2 for unwinding a cDFA C into a counter-free DFA D in an off-line pre-processing step, so that the violation algorithms in Section 2 can be re-used. The queue Q of pairs made of a state of C and corresponding tuple of counter values that still need to be unwound is initialised in line 2 to be empty. Lines 3 and 4 enqueue the initial state s and initial counter values c of C into Q. A partial map m is maintained to map pairs of states and counter values of C to states in D: the pair $\langle s, c \rangle$ is mapped to the initial state 1 of D in lines 5 and 6. The current number n of states of D is initialised to 1 in line 7. In lines 8 to 20, each time an element can be dequeued from Q, newly encountered states and counter values are enqueued for future unwinding, while new states and transitions are added to

Algorithm 2 Unwinding a counter automaton C into a DFA D.

```
1: procedure unwinder(C, D)
2: Q ← [ ]
3: ⟨s, c⟩ ← the initial state and counter values of C
4: Q.enqueue(⟨s, c⟩)
5: m(s, c) ← 1
6: create state 1 as the initial state of D
7: n ← 1
8: while Q ≠ [ ] do
9:    ⟨s, c⟩ ← Q.dequeue()
10:   if state s is a final state of C and its guard holds for counter values c then
11:      mark m(s, c) as a final state of D
12:   for all outgoing transitions t from state s in C do
13:      if the guard of t holds for counter values c then
14:         s′ ← the target state of t in C
15:         c′ ← the values of c after executing the counter assignments of t
16:         if m(s′, c′) is still undefined then
17:            n ← n + 1
18:            m(s′, c′) ← n
19:            Q.enqueue(⟨s′, c′⟩)
20:         add a transition from m(s, c) to m(s′, c′) on the symbol of t to D
```

D. Line 10 checks whether the dequeued state with its counter values is a final state of C; if yes, then the corresponding state is marked as a final state of D in line 11. Lines 12 to 20 examine all outgoing transitions from the dequeued state: if a guarded transition can fire in C, then its counterpart is added to D after enqueuing the corresponding target state if it has not been encountered yet, as in lines 16 to 19.

Example 3.2. Take the cDFA of Figure 3 where the parameters $\langle \underline{d}, \overline{d}, \underline{e}, \overline{e}, \underline{x}, \overline{x} \rangle$ are set to $\langle 2, 2, 1, 2, 1, 2 \rangle$. Its initial state is state 1 with counter c having value 0: the term $\langle 1, 0 \rangle$ is enqueued into the empty queue, and state $(1, 0)$ is set to be the initial state of the result DFA. Next, term $\langle 1, 0 \rangle$ is dequeued, making the queue empty again: all the three unguarded transitions from state 1 can fire and reach new states, so D, X, and E are enqueued with counter value 1, yielding $[\langle D, 1 \rangle, \langle X, 1 \rangle, \langle E, 1 \rangle]$, and three states $(D, 1)$, $(X, 1)$, and $(E, 1)$ are added to the result DFA with transitions from $(1, 0)$ on d, x, and e respectively. Next, D with counter value 1 is dequeued: only the guarded transition $d\ \left\{c < \overline{d} \rightarrow c := c + 1\right\}$ from D can fire (because $c = 1 < 2 = \overline{d}$) and reach a new state, so D with counter value 2 is enqueued, yielding $[\langle X, 1 \rangle, \langle E, 1 \rangle, \langle D, 2 \rangle]$, and state $(D, 2)$ is added to the result DFA with a transition from $(D, 1)$ on d. Next, X with counter value 1 is dequeued: all the three guarded transitions from X can fire, but only state X with counter value 2 has not been encountered before and is enqueued, yielding $[\langle E, 1 \rangle, \langle D, 2 \rangle, \langle X, 2 \rangle]$, and state $(X, 2)$ is added to the result DFA with a transition from $(X, 1)$ on x, as well as a transition from $(X, 1)$ on d to $(D, 1)$ and a transition from $(X, 1)$ on e to $(E, 1)$. Next, E with counter value 1 is dequeued: both the two guarded transitions from E can fire, but only state E with counter value 2 has not been encountered before and is enqueued, yielding $[\langle D, 2 \rangle, \langle X, 2 \rangle, \langle E, 2 \rangle]$, and state $(E, 2)$ is added to the result DFA with a transition from $(E, 1)$ on e, as well as a transition from

$(E, 1)$ on x to $(X, 1)$. Next, D, X, and E with counter value 2 are dequeued one by one, but none of them lead to any unencountered states, and the queue becomes empty, but transitions from $(D, 2)$ and $(E, 2)$ on x to $(X, 1)$ as well as from $(X, 2)$ on d to $(D, 1)$ and from $(X, 2)$ on e to $(E, 1)$ are added. The states $(1, 0)$, $(D, 2)$, $(X, 1)$, $(X, 2)$, $(E, 1)$, and $(E, 2)$ are marked as final states. If we minimise the resulting DFA, then states $(D, 2)$ and $(E, 2)$ are found to be equivalent and are thus merged: the resulting DFA is the one of Figure 1, with states $(1, 0)$, $(D, 1)$, $(X, 1)$, $(E, 1)$, $(D, 2) \equiv (E, 2)$, and $(X, 2)$ corresponding respectively to states 1, 2, 3, 4, 5, and 6.

As each possible combination of states and counter values in the cDFA is enqueued at most once, the worst-case number of states of the resulting DFA is the maximum number of combinations of states and counter values in the cDFA. If the cDFA has m states and c counters, and if r is the number of different values reachable by the counters, then the worst-case number of states of the resulting DFA is $O(m \cdot r^c)$. In the cDFA of Figure 3, the value of r can be determined by inspection to be $\max(\overline{d}, \overline{e}, \overline{x}) + 1 = 3$.

3.3. DFS on Counter Automata

The process of unwinding and unrolling can be done dynamically on demand, as in Section 2.3, for a given number of variables. We modify Algorithm 2 and combine it with Algorithm 1. We again incrementally maintain a partial map m that maps pairs of states and counter values to already explored unwound states. The goal, as in Algorithm 1, is to build a segmentation of the current assignment. Remember that in Algorithm 1 there are two possible choices for each variable V_i with value d_i in the current assignment. Either there is a corresponding transition in the unwound automaton labelled with d_i or there is no corresponding transition and a random transition is picked. Each considered transition has to be checked if there is a path to a success state of the automaton. To check if there is a path from the current node to a success state, a DFS is performed, again updating the map m so as not to reexamine needlessly already unwound states.

In the worst case, this algorithm will unwind the whole automaton, as in Algorithm 2 and for a given number of variables, but the cost will be amortised over many calls to the algorithm.

4. Experiments

We now investigate the practicality of the proposed *automaton* constraint by comparing the designed violation algorithms to each other as well as to the ones of the *regular* constraint [12] for CBLS and handcrafted special-purpose global constraints. Note that the objective of our experiments is thus *not* to beat the state of the art of the considered benchmarks. Indeed, since we compare our general-purpose violation algorithms with hand-crafted special-purpose ones, our purpose is to show that (expensive) human modelling time can be decreased drastically when a missing global constraint is needed, or when one wants to study the impact of conjoining several global constraints that act on the same variables, without getting too high an overhead in (cheap) computer solving time. If extra solving efficiency is required after figuring out a good model and search procedure that use the new global constraint, say because the cost of hand-crafting special-purpose violation algorithms can be amortised over many runs, then one can always hand-craft those algorithms. Our main objective with this work is the rapid prototyping of new global constraints.

The choice of the heuristic and meta-heuristic for a search procedure is an entirely *orthogonal* issue to our concern for the constraint model of a combinatorial problem and for the violation algorithms of its constraints. Hence, in our experiments, we often do not spend much effort on designing a particularly good search procedure, since the *same* search procedure will be applied to all ways of implementing the constraints. One could even argue that a poorly designed search procedure gives a better approximation of the worst-case overhead of using our $automaton$ generic constraint instead of a hand-crafted special-purpose constraint, as there will be an unusually high number of move evaluations and executions.

The experiments are also intended to highlight the modelling convenience of automata, especially of the more compact and generic counter-automata. In particular, the composition of several global constraints (even built-in ones) often becomes straightforward (contrary to the composition of violation algorithms) and often yields better performance than when using the individual constraints. This is especially useful when new constraints are added to a problem.

We have implemented all our algorithms and models in COMET [13], which is an object-oriented constraint programming language with a constraint-based local search back-end (available at `dynadec.com`). We have re-implemented the $regular$ constraint of [12], as its source code is reportedly not available.

All experiments were run under COMET (version 2.0-1) and Mac OS X 10.6.2 on a 2.8 GHz Intel Core 2 Duo with a 4GB RAM. All runs in Sections 4.1 and 4.2 were allocated 30 CPU seconds, and the runs in Section 4.3 were allocated 600 CPU seconds. The average performance was recorded for each instance over 25 runs, starting from the same initial assignments for all the ways of posting the global constraints of a model. In each result table, each row first specifies a (possibly singleton) set of known satisfiable instances, and then gives the performance of each way of posting the global constraints, namely the average percentage of instances solved without timing out (denoted by %S), the average runtime in seconds (denoted by Sec), and the average number of iterations (denoted by Iter).

The experiments were made for the construction of rotating schedules (see Sections 4.1), nurse rostering (see Section 4.2), and car sequencing (see Section 4.3). All discussed global constraints are described in the *Global Constraint Catalogue* [3], where references to their origins can be found.

4.1. Rotating Nurse Schedules

Many industries and services need to function around the clock. Rotating schedules such as the one in Table 1 (a real-life example taken from [9]) are a popular way of guaranteeing a maximum of equity to t work teams (see [9]). There are day (d), evening (e), and night (n) shifts of work, as well as days off (x). Each team works maximum one shift per day. The scheduling horizon has as many weeks as there are teams. In the first week, team i is assigned to the schedule in row i. For any next week, each team moves down to the next row, while the team on the last row moves up to the first row. Note how this gives almost full equity to the teams, except, for instance, that team 1 does not enjoy the six consecutive days off that the other teams have, but rather three consecutive days off at the beginning of week 1 and another three at the end of week 5. The daily workload may be uniform: for instance, in Table 1, each day has exactly one team on-duty for each work shift, and two teams off-duty; we denote this uniform daily workload by $(1d, 1e, 1n, 2x)$; assuming the work shifts average 8h, each employee will work $7 \cdot 3 \cdot 8 = 168$h over the five-week-cycle, or 33.6h per week.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	x	x	x	d	d	d	d
2	x	x	e	e	e	x	x
3	d	d	d	x	x	e	e
4	e	e	x	x	n	n	n
5	n	n	n	n	x	x	x

Table 1. Five-week rotating schedule with uniform daily workload $(1d, 1e, 1n, 2x)$

4.1.1. The Model

Daily workload, whether uniform or not, can be enforced by global cardinality (gcc) constraints on the columns; however, our model does not include those gcc constraints, because of the search procedure (discussed below). In our problem instances, any number of consecutive workdays must be between two and seven, and any transition in work shift can only occur after two to seven days off. The shift transition constraint can be enforced by a $pattern(X, \{(d,x),(e,x),(n,x),(x,d),(x,e),(x,n)\})$ constraint (where a stretch of value d must be followed by a stretch of value x; similarly for stretches of value e or n; a stretch of value x can be followed by a stretch of value d, e, or n) on the table flattened row-wise into a sequence $X[1,\ldots,7{\cdot}t]$ of variables, together with the constraint $X[1] = X[7{\cdot}t] \vee X[1] = x \vee X[7{\cdot}t] = x$, which enforces that the transition between the last and first elements of X is legal. The shift length constraint can be modelled by a $stretch_circular(X, [d,e,n,x], [2,2,2,2], [7,7,7,7])$ constraint (saying that any stretch must have at least 2 and at most 7 elements), with X seen as a circular array. As we will need the related $stretch_path$ constraint also in Section 4.2, we omit modelling the required $stretch_circular$ constraint and approximate it by a $stretch_path$ constraint (with the same arguments) together with the symmetry-breaking constraint $X[1] \neq X[7 \cdot t]$ enforcing that the first and last stretches are over different values.

4.1.2. The Global Constraints

COMET does not have the $pattern$ and $stretch_path$ global constraints as built-ins. Figure 4 gives a DFA for the mentioned $pattern(X, \{(d,x),(e,x),(n,x),(x,d),(x,e),(x,n)\})$ constraint. Figure 5 gives a cDFA for any $pattern(X, P)$ constraint, where a stretch of value v can be followed by a stretch of value w if $(v,w) \in P$, where P contains all the allowed patterns; it describes the same constraint as in Figure 4 when given parameter $P = \{(d,x),(e,x),(n,x),(x,d),(x,e),(x,n)\}$. A guarded arc annotated with "$\forall\gamma \in \Sigma\,\{\alpha \to \beta\}$" in Figure 5 is just a convenience for drawing the counter DFA; it denotes multiple guarded arcs where each arc has a symbol from the alphabet Σ with the same guard α and counter assignment β. Figure 6 gives a DFA for the $stretch_path(X, [d,e,n,x], [2,2,2,2], [7,7,7,7])$ constraint. Figure 7 gives a cDFA for any $stretch_path\ (X, O, \underline{O}, \overline{O})$ constraint: it is much more compact (2 vs 29 states) and generic (the lower bounds need not all be 2, and the upper bounds need not all be 7) than the DFA of Figure 6, so preferable if the bounds are likely to change. Indeed, unwinding the cDFA over $\Sigma = \{d,e,n,x\}$ for $\langle O, \underline{O}, \overline{O}\rangle = \langle [d,e,n,x], [2,2,2,2], [7,7,7,7]\rangle$ with Algorithm 2 and minimising the result gives the DFA in Figure 6. Alternatively, one can construct [3] a DFA directly from the given instance of the $stretch_path$ constraint and minimise it: the resulting DFA is the same as the minimised unwound one, but our unwinding is not specific to the $stretch_path$ constraint and thus more general.

When using $automaton$, our model uses the minimised product of the $pattern$ and $stretch_path$ (counter) automata, accepting the intersection of their two languages. Indeed, this has been determined

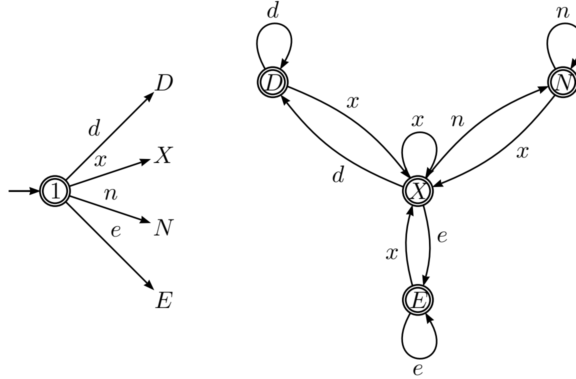

Figure 4. An automaton for the $pattern(X, \{(d,x),(e,x),(n,x),(x,d),(x,e),(x,n)\})$ constraint over four-letter alphabet $\{d,e,n,x\}$ in the chosen instances for rotating nurse scheduling.

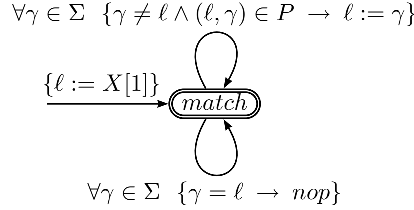

Figure 5. A counter automaton for any $pattern(X, P)$ constraint, where a stretch of value v can be followed by a stretch of value w if $(v,w) \in P$, where P contains all the allowed patterns. The counter ℓ records the value of the current stretch, and is initialised to the value of variable $X[1]$.

by our experiments (not reported here) to be much more efficient than using the two automata individually, no matter which implementation of the *automaton* constraint we deploy. Figure 8 gives the minimised product of the cDFA of Figure 5 and the cDFA of Figure 7: this cDFA is preferable since unwinding it for $\langle O, \underline{O}, \overline{O}, P\rangle = \langle [d,e,n,x],[2,2,2,2],[7,7,7,7],\{(d,x),(e,x),(n,x),(x,d),(x,e),(x,n)\}\rangle$ with Algorithm 2 and minimising the result gives the minimised product of the DFAs in Figures 4 and 6.

When not using *automaton*, our model uses the handcrafted *pattern* and *stretch_path* constraints discussed below, though without combining their violation algorithms, as there currently is no calculus for doing that.

A handcrafted violation algorithm for the present $pattern(X, [d,e,n,x], [\{x\},\{x\},\{x\},\{d,e,n\}])$ constraint instance was quickly designed using the differentiable invariants [14] of COMET, which lift logical expressions into constraints. Indeed, the following formula captures this instance:

$$\forall i \in [2,\ldots,7\cdot t] : X[i-1] \neq X[i] \lor X[i-1] = x \lor X[i] = x$$

4.1.3. Interlude: A Handcrafted *stretch_path* Constraint

A handcrafted violation algorithm for any *stretch_path* constraint instance was designed much more laboriously, after several hours of thinking and experimentation with alternatives.

Consider a *stretch_path* constraint for n variables X_i over m values d_j with lower bounds $\underline{O}[d_j]$ and upper bounds $\overline{O}[d_j]$ on the lengths of stretches. In the proposed Algorithm 3, the local variables s and e

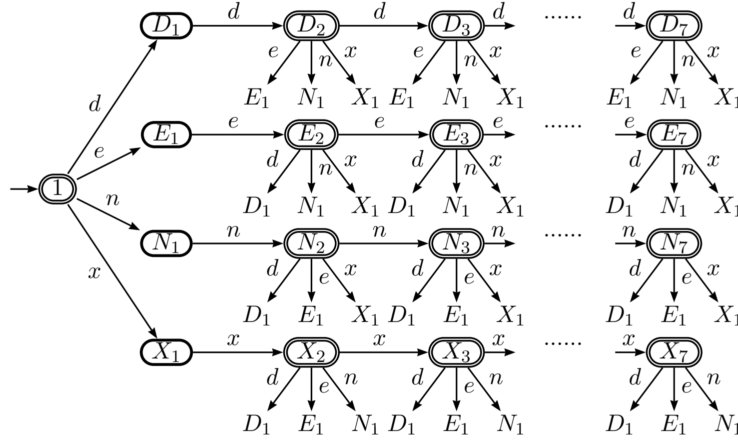

Figure 6. An automaton for the $stretch_path(X, [d, e, n, x], [2, 2, 2, 2], [7, 7, 7, 7])$ constraint over four-letter alphabet $\{d, e, n, x\}$ in the chosen instances for rotating nurse scheduling.

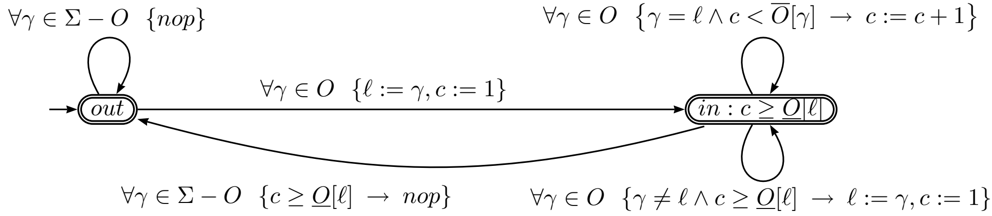

Figure 7. A counter automaton for any $stretch_path(X, O, \underline{O}, \overline{O})$ constraint, each stretch of value $v \in O$ being of a length between lower bound $\underline{O}[v]$ and upper bound $\overline{O}[v]$; the counter ℓ records the value of the current stretch; the counter c maintains the length of the current stretch; most transitions and one final state are guarded by comparisons between c and the length bounds on the current stretch.

respectively record the indices of the first and last variables of the most recent stretch, whose value and length the local variables d and ℓ record. Line 2 initialises s and d for the first variable. Lines 3 and 4 respectively initialise the violation of the constraint and its variables (including dummy variables X_0 and X_{n+1}) to 0. Lines 5 to 17 scan the remaining variables from left to right: each time a stretch ends (when the test in line 6 succeeds), lines 7 and 8 first update e and ℓ, and then a case analysis is performed:

- The violation of the constraint is updated in lines 10 and 16 according to the following formula, when there are s stretches of lengths ℓ_i and values d_i in the current assignment:

$$Violation = \sum_{i=1}^{s} \max(\ell_i - \overline{O}[d_i], 0) + \max(\underline{O}[d_i] - \ell_i, 0)$$

  A too long (line 9) or too short (line 13) stretch contributes the length of its overflow (line 10) or underflow (line 16) to the constraint violation.

- The violations of the variables cannot easily be captured in such an aggregating formula. If a stretch is too long (line 9), then its first and last variables are considered to be violated and have

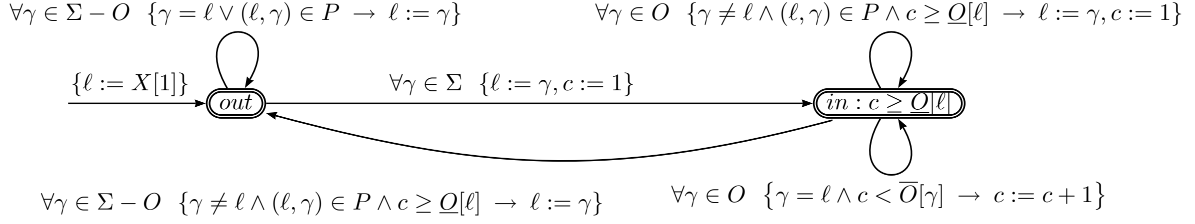

Figure 8. (Conjunctive) product of the *pattern* and *stretch_path* counter automata of Figures 5 and 7.

their violations incremented by one (lines 11 and 12), so that the next move might change one of those variables and thus possibly decrease the length of the stretch by one; a stretch overflowing by more than one value can thus be trimmed to a suitable size in several moves. Conversely, if a stretch is strictly shorter than allowed (line 13), then its preceding and succeeding variables (which are possibly the dummy variables created in line 4) are considered to be violated and have their violations incremented by one (line 14), so that the next move might change one of those variables and thus possibly increase the length of the stretch by one; a stretch underflowing by more than one value can thus be extended to a suitable size in several moves.

We argue that it is not a good idea only to increase the length of too short stretches, as we can also decrease their length to eliminate them (that is, to add line 15 in Algorithm 3, making the first and last variables of a too short stretch violated). Consider a *stretch_path* constraint over $\Sigma = \{d, e, x\}$ with parameters $\langle O, \underline{O}, \overline{O}\rangle = \langle [d, e, x], [2, 2, 2], [3, 3, 3]\rangle$, and values $\langle d, d, e, x, x\rangle$ for a sequence $\langle X_1, \dots, X_5\rangle$ of 5 variables. It has three stretches: $s_1 = \langle d, d\rangle$, $s_2 = \langle e\rangle$, and $s_3 = \langle x, x\rangle$. As s_2 is the only stretch that makes the constraint violated and as it is a too short stretch of value e, the only way to enlarge s_2 is to change variable X_2 to e, or to change X_4 to e. If the move $X_2 := e$ is chosen, then $s_1 = \langle d\rangle$ is too short. In order to enlarge s_1 then, variable X_2 may be changed back to d, and $s_2 = \langle e\rangle$ becomes too short again. Under this situation, enlarging a too short stretch will always make another stretch too short. However, instead of enlarging s_2, eliminating it by changing X_2 to d or x (since stretches s_1 and s_3 are shorter than allowed) will make the constraint satisfied.

In practice, we use an incremental version of this algorithm, maintaining the values of s, e, d, and ℓ for every stretch.

Example 4.1. Consider the $stretch_path([X_1, \dots, X_{10}], [d, e, x], [2, 2, 4], [3, 4, 5])$ constraint over $\Sigma = \{d, e, x\}$. Under the sequence $\langle d, d, d, x, e, e, d, d, d, d\rangle$ of values, the violations of the constraint and all variables are initialised to 0. The first stretch, of value d, is of maximal length, so it does not affect any violations. The singleton stretch of value x is too short, so it increases the constraint violation to 1; as is it is shorter than allowed, the violations of its neighbour variables X_3 and X_5 are increased to 1; as its first and last variables are the same (variable X_4), so the violation of X_4 are increased to 2. The stretch of value e is of allowed length, so it does not affect the constraint violation. The last stretch, of value d, is too long, so it increases the constraint violation to 2, and it increases the violations of its first and last variables X_7 and X_{10} to 1. In summary, the constraint has violation 2 and the most violated variable is X_4 with also violation 2. The move $X_4 := e$ will decrease the constraint violation to 1 (because of the still too long stretch of d), decrease the violations of X_3, X_4 and X_5 to 0.

Algorithm 3 Handcrafted violation algorithm for the $stretch_path$ constraint

1: **procedure** $calcViolations(stretch_path([X_1, \dots, X_n], O, \underline{O}, \overline{O}))$
2: $s \leftarrow 1; d \leftarrow X_1$
3: $Violation \leftarrow 0$
4: **for all** $i = 0$ **to** $n + 1$ **do** $Violation[X_i] \leftarrow 0$ **end for**
5: **for all** $i = 2$ **to** n **do**
6:   **if** $X_i \neq d$ **then**
7:     $e \leftarrow i - 1$
8:     $\ell \leftarrow e - s + 1$
9:     **if** $\ell > \overline{O}[d]$ **then**
10:       $Violation \leftarrow Violation + (\ell - \overline{O}[d])$
11:       $Violation[X_s] \leftarrow Violation[X_s] + 1$
12:       $Violation[X_e] \leftarrow Violation[X_e] + 1$
13:     **else if** $\ell < \underline{O}[d]$ **then**
14:       $Violation[X_{s-1}] \leftarrow Violation[X_{s-1}] + 1$; $Violation[X_{e+1}] \leftarrow Violation[X_{e+1}] + 1$
15:       $Violation[X_s] \leftarrow Violation[X_s] + 1$; $Violation[X_e] \leftarrow Violation[X_e] + 1$
16:       $Violation \leftarrow Violation + (\underline{O}[d] - \ell)$
17:     $s \leftarrow i; d \leftarrow X_i$ {start a new stretch}

4.1.4. The Search Procedure

A gcc constraint can be maintained by a neighbourhood containing only assignments that swap the values of two of its variables. It is often much more efficient to enforce a gcc constraint with such a neighbourhood than to post it as a constraint in the model, using a built-in constraint, a DFA, or a cDFA. This is the case here (comparisons are not given here for space reasons), so we omit the gcc constraints from the model. Recall that Algorithm 1 greedily computes a segmentation under random choices when greedy segment growth is impossible. Hence it might give different segmentations when probing a swap and when actually performing that swap. Therefore, we record the segmentation of each swap probe, and at the actual swap, we just apply its recorded segmentation. Upon uniform workload, we can deterministically construct a suitable initial assignment, which experiments (reported in [6]) have shown to be much better than random initial assignments. For example, the initial assignment of instance $(16d, 8e, 8n, 16x)$ consists of 8 vertically stacked copies of the initial assignment in Table 2 for instance $(2d, 1e, 1n, 2x)$, which itself was systematically obtained by satisfying the gcc constraint in each column and the $stretch_path$ constraint in each row. We apply min-conflict on the most violated variable, because this works well with the meta-heuristic discussed next.

Our chosen meta-heuristic is tabu search with restarts. At each iteration, the search procedure selects a violated variable x (recall that the violation of a variable is here at most 1) and another variable y of distinct value in the same column so that their swap gives the greatest violation change. The length of the tabu list is the maximum between 6 and the sum of the violations of all constraints. The best solution so far is maintained. Restarting is done every $2 \cdot |X|$ iterations. The expressions for the length of the tabu list and the restart criterion were experimentally determined.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	d	d	d	d	d	d	d
2	e	e	e	e	e	e	e
3	n	n	n	n	n	n	n
4	x	x	x	x	x	x	x
5	d	d	d	d	d	d	d
6	x	x	x	x	x	x	x

Table 2. Non-random initial assignment for the rotating nurse schedule with uniform daily workload $(2d, 1e, 1n, 2x)$

4.1.5. Results

We ran experiments over the eight satisfiable instances with uniform daily workload $(1d, 1e, 1n, 1x)$ until $(8d, 8e, 8n, 8x)$, the latter being denoted by $(1d, 1e, 1n, 1x) \cdot 8$, and over the eight satisfiable instances with uniform daily workload $(2d, 1e, 1n, 2x) \cdot 1$ until $(2d, 1e, 1n, 2x) \cdot 8$. Table 3 compares the discussed five ways of implementing the $pattern$ and $stretch_path$ constraints. We observe that all the $automaton$ implementations can solve the instances $(1d, 1e, 1n, 1x) \cdot 1$ to $(1d, 1e, 1n, 1x) \cdot 8$ more efficiently than the handcrafted $pattern$ and $stretch_path$ constraints; for instances $(2d, 1e, 1n, 2x) \cdot 1$ to $(2d, 1e, 1n, 2x) \cdot 8$, the $regular$ constraint [12] is less efficient than the other four implementations, which have similar runtimes. This shows the possibility that a generic constraint can even beat a carefully designed handcrafted constraint. This is because it combines the $pattern$ and $stretch_path$ constraints into one constraint, and gives a better estimate of violations. Our three $automaton$ implementations have close runtimes, and there is no clear winner. Compared with $regular$ [12], our methods tend to have a higher number of iterations, as they are more stochastic; however, our runtimes are lower, as our cost of one iteration is much smaller (linear in the number of variables, instead of linear in the number of arcs of the unrolled automaton).

4.2. The Nurse Scheduling Problem

NSPlib [19] is a very large repository of (artificially generated) instances of the *nurse scheduling problem* (NSP), which is about constructing a duty roster for nursing staff. Let N be the number of nurses, D the number of days of the scheduling horizon, and S the number of shifts. The objective is to construct an $N \times D$ matrix of values in the integer interval $[1, \ldots, S]$, with value S representing the off-duty "shift".

4.2.1. The Model

In *instance files*, there are hard *coverage constraints* and soft preference constraints; we only use the former here (as optimisation is orthogonal to our modelling concerns, so that we want to keep the instances solvable within a short amount of time): they give for each day d and shift s the lower bound on the number of nurses that must be assigned to shift s on day d, and can be modelled by $atLeast$ constraints on the columns. There are instance files for $N \times 7$ rosters with $N \in \{25, 50, 75, 100\}$, and for $N \times 28$ rosters with $N \in \{30, 60\}$.

In *case files*, there are hard constraints on the rows. For each shift s, there are lower and upper bounds on the number of occurrences of s in any row (the daily assignment of some nurse): this can be modelled by gcc constraints on the rows. There are even lower and upper bounds on the cumulative number of

	Unrolled DFA			DFS on DFA			DFS on cDFA			[12] on DFA			Handcrafted		
Instance	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter
$(1d, 1e, 1n, 1x) \cdot 1$	100	0.004	25	100	0.004	26	100	0.003	22	100	0.03	23	100	0.018	174
$(1d, 1e, 1n, 1x) \cdot 2$	100	0.008	19	100	0.005	18	100	0.005	20	100	0.051	17	100	0.061	504
$(1d, 1e, 1n, 1x) \cdot 3$	100	0.014	34	100	0.014	49	100	0.011	31	100	0.176	47	100	0.130	925
$(1d, 1e, 1n, 1x) \cdot 4$	100	0.019	48	100	0.027	73	100	0.02	47	100	0.215	43	100	0.338	2113
$(1d, 1e, 1n, 1x) \cdot 5$	100	0.027	66	100	0.033	61	100	0.033	65	100	0.303	50	100	0.494	2636
$(1d, 1e, 1n, 1x) \cdot 6$	100	0.037	76	100	0.049	85	100	0.048	79	100	0.481	67	100	0.988	4704
$(1d, 1e, 1n, 1x) \cdot 7$	100	0.049	130	100	0.061	90	100	0.068	99	100	0.625	77	100	1.605	6812
$(1d, 1e, 1n, 1x) \cdot 8$	100	0.062	110	100	0.126	175	100	0.116	155	100	0.830	100	0	1.481	5691
$(1d, 1e, 1n, 1x)$	100	0.028	64	100	0.040	72	100	0.038	65	100	0.339	52	100	0.639	2945
$(2d, 1e, 1n, 2x) \cdot 1$	100	0.005	8	100	0.001	9	100	0.003	13	100	0.031	9	100	0.010	42
$(2d, 1e, 1n, 2x) \cdot 2$	100	0.019	21	100	0.009	24	100	0.011	31	100	0.091	22	100	0.026	108
$(2d, 1e, 1n, 2x) \cdot 3$	100	0.029	41	100	0.029	63	100	0.024	50	100	0.223	38	100	0.041	100
$(2d, 1e, 1n, 2x) \cdot 4$	100	0.033	78	100	0.055	96	100	0.036	56	100	0.426	57	100	0.074	128
$(2d, 1e, 1n, 2x) \cdot 5$	100	0.064	142	100	0.082	107	100	0.073	96	100	0.714	80	100	0.088	167
$(2d, 1e, 1n, 2x) \cdot 6$	100	0.076	126	100	0.119	136	100	0.104	112	100	1.051	100	100	0.128	214
$(2d, 1e, 1n, 2x) \cdot 7$	100	0.096	156	100	0.205	211	100	0.19	185	100	1.449	119	100	0.128	222
$(2d, 1e, 1n, 2x) \cdot 8$	100	0.134	235	100	0.208	178	100	0.307	270	100	2.157	157	100	0.182	280
$(2d, 1e, 1n, 2x)$	100	0.057	101	100	0.089	103	100	0.094	102	100	0.768	73	100	0.085	158

Table 3. Benchmark results on rotating nurse schedules

occurrences of the working shifts $1, \dots, S-1$ in any row: this can be modelled by gcc constraints on the off-duty value S and always gives tighter occurrence bounds on S than in the previous gcc constraints. For each shift s, there are also lower and upper bounds on the length of any stretch of value s in any row: this can be modelled by $stretch_path$ constraints on the rows. Finally, there are lower and upper bounds on the length of any stretch of the working shifts $1, \dots, S-1$ in any row: this can be modelled by $stretch_path_partition$ constraints on the rows. We stress that the constraints on any two rows are the *same*. There are 8 case files for the $N \times 7$ rosters, and another 8 case files for the $N \times 28$ rosters.

4.2.2. The Global Constraints

COMET does not have the $stretch_path$ and $stretch_path_partition$ global constraints as built-ins. (Counter) automata for $stretch_path$ were discussed in Section 4.1. A DFA for the $stretch_path_partition(X, [\{d, e, n\}], [2], [4])$ constraint of NSPlib Case 8 is given in Figure 9. A cDFA (not given here for space reasons) for any $stretch_path_partition(X, [\{d, e, n\}], [\ell], [u])$ constraint over an alphabet containing $\{d, e, n\}$ trivially generalises the cDFA of Figure 7 for $stretch_path$. A handcrafted violation algorithm (not given here for space reasons) for the $stretch_path_partition$ constraint was designed by trivial generalisation of our handcrafted Algorithm 3 for the $stretch_path$ constraint.

COMET does not have a built-in gcc constraint, but it can be simulated, without loss of efficiency compared to a handcrafted implementation, by conjoining the $atLeast$ and $atMost$ built-in constraints of COMET. We have constructed DFAs (not given here for space reasons, as they would be huge) for the case-specific instances of the gcc constraint. We have also designed a cDFA (not given here for space reasons)for any instance of the gcc constraint. Unwinding this cDFA for any instance with Algorithm 2 and minimising the result gives the same DFA as our specific construction for that instance.

Our model uses the $atLeast$ built-in constraint of COMET on the columns of the duty roster.

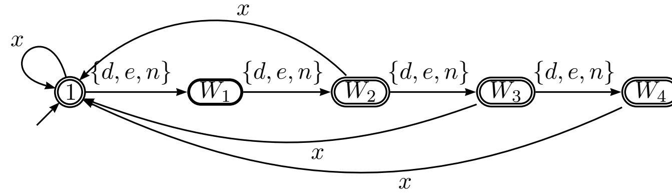

Figure 9. An automaton for the $stretch_path_partition((X, [\{d, e, n\}], [2], [4]))$ constraint of Case 8 of NSPlib.

When using $automaton$, our model actually uses the minimised unwound product of the $stretch_path$, $stretch_path_partition$, and gcc cDFAs, accepting the intersection of their three languages. Indeed, this has been determined by our experiments (not reported here) to be more efficient than using the three automata individually, no matter which implementation of the $automaton$ constraint we deploy.

When not using $automaton$, our model uses our handcrafted $stretch_path$ and $stretch_path_partition$ constraints as well as the built-in gcc constraints on the rows of the duty roster, though without combining their violation algorithms, as there currently is no calculus for doing that.

4.2.3. The Search Procedure

Our chosen heuristic is min-conflict on the most violated variable, starting from a random initial assignment. As modifying the assignment of the most violated variable often helps to find a better solution, and as only a small neighbourhood is searched in each iteration, this search heuristic is often successful. This is the case here. We only consider assignment moves. Our chosen meta-heuristic is tabu search with restarts, under the same settings as in Section 4.3.

4.2.4. Results

For each case and nurse count N, we used the ***first*** 10 instances for each configuration of the NSPlib coverage complexity indicators, that is instances 1–270 for the $N \times 7$ rosters and 1–120 for the $N \times 28$ rosters. We restricted ourselves to the instances shown to be satisfiable by a (propagation-based) constraint program within one CPU minute on the same hardware.

Table 4 compares the discussed five ways of implementing the row constraints on the chosen satisfiable NSPlib instances of Cases 7 and 8 only (for space reasons). We observe that the handcrafted built-in constraints are more efficient than the $automaton$ implementations thereof, but not by a wide margin on Case 8.

4.3. Car Sequencing

The car sequencing problem consists of sequencing the production of n cars of the same basic model, but with possibly different options (air-conditioning, sun-roof, etc) installed, so that the capacity constraints of the stations on the assembly line are never exceeded. These capacity constraints are of the form u/q, meaning that at most u out of any q successive cars can have a particular option installed at some station. All the considered problem instances have the same capacity constraints, namely $1/2$, $2/3$, $1/3$, $2/5$, and $1/5$ for five options; the instances differ in the demand constraints, that is the numbers of cars of each configuration of the five options that are to be assembled.

		Unrolled DFA			DFS on DFA			DFS on cDFA			[12] on DFA			Handcrafted		
Case	N	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter
7	25	73	9.298	154418	55	15.023	141283	55	15.013	129571	56	15.863	64987	88	4.206	44140
7	50	47	17.111	155676	37	19.943	122096	37	20.052	115776	11	28.099	87053	81	6.432	43548
7	75	32	21.207	152183	26	22.915	121479	26	22.991	111138	1	29.900	91029	77	8.150	46027
7	100	36	20.433	107148	33	21.316	84683	33	21.457	84521	0	30.044	75893	75	8.869	39479
8	25	95	2.001	31377	67	12.690	102612	65	13.197	95384	92	3.906	15307	91	3.412	33074
8	50	78	7.224	66941	55	14.427	106321	53	15.521	98974	71	11.415	36073	79	6.957	47984
8	75	80	7.173	52538	27	22.958	120967	24	23.766	75559	57	17.786	56597	81	6.648	39160
8	100	73	9.347	48841	24	24.093	94171	23	24.212	86653	19	26.959	73021	76	8.181	35905

Table 4. Benchmark results on NSPlib instances

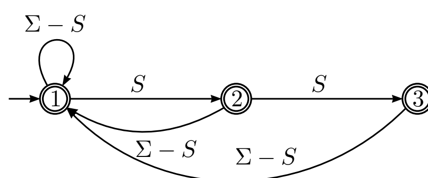

Figure 10. An automaton for the $sequence(X, 0, 2, 3, S)$ constraint used in instances of the car sequencing problem, where Σ is the alphabet.

4.3.1. The Model

This problem can be modelled by an array X of n variables, with five $sequence(X, \ell, u, q, S)$ constraints, meaning that between ℓ and u of any q successive variables in X take a value in the set S (here a set of options), with $0 \leq \ell \leq u \leq q$ and $1 \leq q \leq n$. Each of the five *sequence* constraints is weighted by a factor indicating how heavily utilised the corresponding option is (see [13] for details). The demand constraints can be enforced by one global cardinality (*gcc*) constraint on X; however, our model does not include that *gcc* constraint, because of the search procedure (discussed below).

4.3.2. The Global Constraints

COMET does have a built-in *sequence* constraint, which actually omits the lower bound ℓ (always 0 here); its violation algorithm is fully described in [13].

A DFA for the $2/3$ capacity constraint, that is the $sequence(X, 0, 2, 3, S)$ constraint, is given in Figure 10. A cDFA for any $sequence(X, \ell, u, q, S)$ constraint is given in Figure 11. Unwinding this cDFA for $\langle \ell, u, q \rangle = \langle 0, 2, 3 \rangle$ with Algorithm 2 and minimising the result gives the DFA in Figure 10. Alternatively, but only in case $S = \{1\}$ and $\Sigma = \{0, 1\}$, one can construct [18] a DFA directly from an instance of the *sequence* constraint and minimise it: the resulting DFA is the same as our minimised unwound one, but our unwinding is not specific to the *sequence* constraint and thus more general.

4.3.3. The Search Procedure

We use the very efficient heuristic (min-conflict on the most violated variable, starting from a random permutation that satisfies the demand constraints, and maintaining those constraints via swap moves) and meta-heuristic (tabu search with intensification, diversification, and restarts) given in [13].

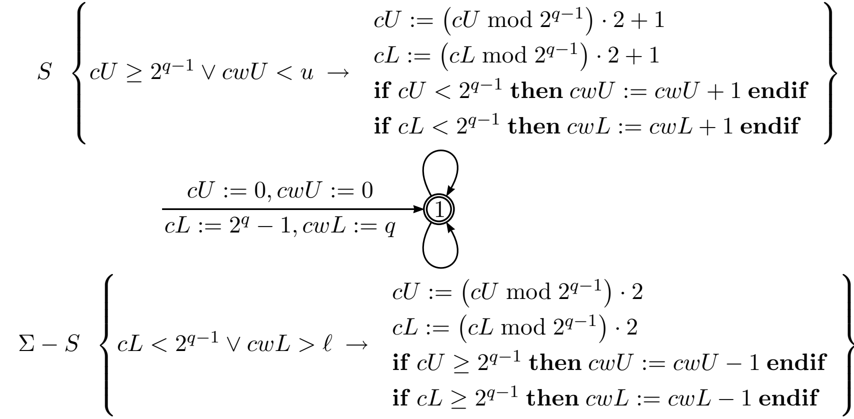

Figure 11. A counter automaton for the $sequence(X, \ell, u, q, S)$ constraint, where Σ is the alphabet. The binary representation of counter cU, denoted by cU_2, gives the content of the current window of length q that we slide across X: bit i of cU_2 is 1 if and only if symbol i in the current window is an element of S. Further, cwU records the number of symbols of S in the current window. The guard $cU \geq 2^{q-1} \vee cwU < u$ tests whether another symbol of S can be read under any of the following two situations: (1) the first symbol of the current window belongs to S; (2) the number of symbols of S in the current window is smaller than u. The counter operation $cU := (cU \bmod 2^{q-1}) \cdot 2 + 1$ updates cU for the next window; similarly cwU is updated. In initialisation, both of cU and cwL are 0 , which corresponds to an initial dummy window to the left of X with no symbols of S. However, this initial window does not satisfy the lower-bound requirement of the constraint, so another two counters cL and cwL are used for the lower bound. cL is initialised to $2^q - 1$, and cwL is initialised to q, which corresponds to another initial dummy window, with only symbols of S. They have a similar guarded transition to cU and cwU.

4.3.4. Results

Table 5 compares the discussed five ways of implementing the *sequence* constraint on the instance of [5], as well as on the ten 60–x instances and the first four satisfiable x/y instances at `CSPlib.org`. We observe that the built-in *sequence* constraint is much more efficient and successful than the *automaton* implementations thereof. On the hard instances, the *automaton* implementations fail to find a solution for most of the 25 runs; on the easy instances, the runtimes are close, though. All this is not surprising because the handcrafted violation algorithm [13] (for the upper-bound part of the) *sequence* constraint is very natural but is not approximated in any sense by the general-purpose violation algorithm of our *automaton* constraint. We included this benchmark as an indicator that the gap between generality and specificity can be very large.

5. Conclusion

In summary, we have shown that the idea of describing novel constraints by (counter) automata can be successfully imported from classical (propagation-based global search) constraint programming to constraint-based local search (CBLS). Our violation algorithms take time linear in the number of variables, whereas the propagation algorithms take amortised time linear in the number of arcs of the unrolled automaton [2, 11]. We have experimentally shown that our approach is competitive with the prior CBLS approach of [12].

	Unrolled DFA			DFS on DFA			DFS on cDFA			[12] on DFA			Built-in Sequence		
Instance	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter
[5]	100	0.015	37	100	0.036	34	100	0.048	39	100	0.099	109	100	0.010	32
60–01…10	100	0.859	130	100	2.646	120	100	2.638	109	100	2.484	126	100	0.204	62
4/72	32	494	145434	4	583	51849	8	566	45860	16	571	57450	100	18.147	133473
16/81	0	600	174299	0	600	52979	4	591	46353	0	600	59400	80	153	1090803
41/66	60	295	94918	28	465	42973	60	316	27069	24	470	49219	100	3.014	23860
28/82	40	475	144873	20	565	51169	20	539	44958	16	551	56874	100	1.862	13113

Table 5. Benchmark results on car sequencing

There is of course a trade-off between using a (counter) automaton to describe a constraint and using a handcrafted implementation of that constraint. On the one hand, a handcrafted implementation of a constraint is normally more efficient, because properties of the constraint can be exploited, but it may take a lot of time to implement and verify it. On the other hand, the (violation or propagation) algorithm processing a automaton is implemented and verified once and for all, and our assumption is that it takes a lot less time to describe and verify a new constraint by an automaton than to implement and verify its algorithm. We see thus opportunities for rapid prototyping with constraints described by automata: once a sufficiently efficient model and search procedure have been experimentally determined with its help, some extra efficiency may be achieved, if necessary, by handcrafting implementations of any constraints described by automata.

As witnessed by our experiments, constraint composition (by conjunction) is easy to experiment with under the automaton approach, as there exist standard and efficient algorithms for composing and minimising automata, but there is no known systematic way of composing violation (or propagation) algorithms when decomposition is believed to obstruct efficiency.

In the global search approach to CP, the common modelling device of reification can be used to shrink the size of DFAs describing constraints [2]. For instance, consider the $element([x_1, \ldots, x_n], i, v)$ constraint, which holds if and only if $x_i = v$. Upon reifying the variables $x_1, \ldots, x_n$ into new Boolean (0/1) variables $b_1, \ldots, b_n$ such that $x_i = v \Leftrightarrow b_i = 1$, it suffices to pose the $automaton([b_1, \ldots, b_n], A)$ constraint, where automaton A corresponds to the regular expression $0^*1(0+1)^*$, meaning that there must be at least one 1 in the sequence of the b_i variables. However, such explicit reification constraints are not necessary in constraint-based local search, as a total assignment of values to all variables is maintained at all times: instead of processing the *values* of the variables when computing the segments, one can process their *reified values*.

Future work includes investigating an extension to push-down automata in order to handle context-free languages in a manner different from the counter automata of this paper. We also want to investigate a violation measure for the $automaton$ constraint that is not based on Hamming distance (which only allows assignment and swap moves) but, say, on Levenshtein distance, which also allows insertion and deletion moves.

Acknowledgements

The authors are supported by grant 2007-6445 of the Swedish Research Council (VR), and Jun He is also supported by grant 2008-611010 of China Scholarship Council and the National University of Defence Technology of China. Many thanks to Magnus Ågren (SICS) for some useful discussions on this work,

and to the anonymous referees of LSCS'09, of the Doctoral Programme of CP'09, of RCRA'09, and of this version of the paper.

References

[1] Ågren, M., Flener, P., Pearson, J.: Generic incremental algorithms for local search, *Constraints*, **12**(3), September 2007, 293–324.

[2] Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers, *Proceedings of CP'04* (M. Wallace, Ed.), LNCS 3258, Springer-Verlag, 2004, 107–122.

[3] Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global constraint catalogue: Past, present, and future, *Constraints*, **12**(1), March 2007, 21–62, The catalogue is at `http://www.emn.fr/x-info/sdemasse/gccat`.

[4] Cheng, K. C. K., Yap, R. H. C.: Maintaining generalized arc consistency on ad hoc r-ary constraints, *Proceedings of CP'08* (P. J. Stuckey, Ed.), LNCS 5202, Springer-Verlag, 2008, 509–523.

[5] Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming, *Proceedings of ECAI'88* (Y. Kodratoff, Ed.), Pitman, 1988, 290–295.

[6] He, J., Flener, P., Pearson, J.: Toward an *automaton* constraint for local search, *Proceedings of LSCS'09, the 6th International Workshop on Local Search Techniques in Constraint Satisfaction* (Y. Deville, C. Solnon, Eds.), Electronic Proceedings in Theoretical Computer Science 5, 2009, 13–25.

[7] Hopcroft, J. E., Motwani, R., Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation*, 3rd edition, Addison-Wesley, 2007.

[8] Lagerkvist, M. Z.: *Techniques for Efficient Constraint Propagation*, KTH – Royal Institute of Technology, Stockholm, Sweden, November 2008, Licentiate Thesis.

[9] Laporte, G.: The art and science of designing rotating schedules, *Journal of the Operational Research Society*, **50**(10), October 1999, 1011–1017.

[10] Michel, L., Van Hentenryck, P.: Localizer: A modeling language for local search, *Proceedings of CP'97* (G. Smolka, Ed.), LNCS 1330, Springer-Verlag, 1997, 237–251.

[11] Pesant, G.: A regular language membership constraint for finite sequences of variables, *Proceedings of CP'04* (M. Wallace, Ed.), LNCS 3258, Springer-Verlag, 2004, 482–495.

[12] Pralong, B.: *Implémentation de la contrainte* Regular *en Comet*, Master Thesis, École Polytechnique de Montréal, Canada, 2007.

[13] Van Hentenryck, P., Michel, L.: *Constraint-Based Local Search*, The MIT Press, 2005.

[14] Van Hentenryck, P., Michel, L.: Differentiable invariants, *Proceedings of CP'06* (F. Benhamou, Ed.), LNCS 4204, Springer-Verlag, 2006, 604–619.

[15] Van Hentenryck, P., Michel, L., Liu, L.: Constraint-based combinators for local search, *Proceedings of CP'04* (M. Wallace, Ed.), LNCS 3258, Springer-Verlag, 2004, 47–61.

[16] Van Hentenryck, P., Saraswat, V., Deville, Y.: *Design, implementation, and evaluation of the constraint language cc(FD)*, Technical Report CS-93-02, Brown University, Providence, USA, January 1993.

[17] Van Hoeve, W.-J., Pesant, G., Rousseau, L.-M.: On global warming: Flow-based soft global constraints, *Journal of Heuristics*, **12**(4-5), September 2006, 347–373.

[18] Van Hoeve, W.-J., Pesant, G., Rousseau, L.-M., Sabharwal, A.: Revisiting the *sequence* constraint, *Proceedings of CP'06* (F. Benhamou, Ed.), LNCS 4204, Springer-Verlag, 2006, 620–634.

[19] Vanhoucke, M., Maenhout, B.: On the characterization and generation of nurse scheduling problem instances, *European Journal of Operational Research*, **196**(2), 2009, 457–467, NSPLib is at `www.projectmanagement.ugent.be/nsp.php`.

[20] Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics, *Constraints*, **14**(3), 2009, 392–413.