2013 IEEE 25th International Conference on Tools with Artificial Intelligence

Generation of Implied Constraints for Automaton-Induced Decompositions

María Andreína Francisco Rodríguez, Pierre Flener, and Justin Pearson
Department of Information Technology
Uppsala University
Uppsala, Sweden
Email: {Maria.Andreina.Francisco, Pierre.Flener, Justin.Pearson}@it.uu.se

***Abstract*—Automata, possibly with counters, allow many constraints to be expressed in a simple and high-level way. An automaton induces a decomposition into a conjunction of already implemented constraints. Generalised arc consistency is not generally maintained on decompositions induced by counter automata with more than one state or counter. To improve propagation of automaton-induced constraint decompositions, we use automated tools to derive loop invariants from the constraint checker corresponding to the given automaton. These loop invariants correspond to implied constraints, which can be added to the decomposition. We consider two global constraints and derive implied constraints to improve propagation even to the point of maintaining generalised arc consistency.**

***Keywords*-constraint programming; implied constraints; global constraints; generalised arc consistency; invariants; automata**

I. Introduction

In constraint programming (CP), a *global constraint* restricts a non-fixed number of decision variables. For example, the ALLDIFFERENT$(x_1, \ldots, x_n)$ constraint holds if and only if the n decision variables x_i take n distinct values. Global constraints are important components of modern CP solvers. A global constraint does two things: from the modelling perspective, it allows a modeller to express a commonly occurring combinatorial substructure; from the solving perspective, it comes with a propagation algorithm, called a *propagator*, which removes impossible values from the domains of its decision variables when invoked during systematic search. There are global constraints for many combinatorial structures, such as scheduling [1], [2], packing [3], and rostering [4]. For a fairly exhaustive survey, see the *Global Constraint Catalogue* [5].

Although modern constraint solvers have many global constraints, often a constraint that one is looking for is not there. In the past, the choices were either to reformulate the model or to write one's own propagator.

In [6], [7], a framework is given for defining many global constraints in a relatively simple and high-level way by a deterministic finite automaton, possibly with counters (in the case of [6]). Such an automaton corresponds to a constraint checker, expressed as a simple imperative program. Based on the automaton, the framework of [6] decomposes the specified new global constraint into a conjunction of already implemented (global) constraints. This conjunction gives the semantics of the specified global constraint and provides the propagation. Unfortunately, generalised arc consistency (GAC) is in general not maintained on decompositions induced by counter automata with more than one state or more than one counter, which means that when the propagators of the constraints of the decomposition reach a common fixpoint, not all infeasible values have been removed from the domains of the decision variables, even if GAC is maintained individually on those constraints.

In this paper, we investigate deriving loop invariants from constraint checkers and use them as implied constraints to extend the corresponding automaton-induced decomposition in order to improve propagation. To illustrate the process we study two global constraints, namely the JTHNONZEROPOS and NGROUP constraints, and show that propagation is improved. Moreover, in the case of the JTHNONZEROPOS constraint we prove that GAC is maintained by the *extended decomposition*, that is, the automaton-induced decomposition extended with the implied constraints. The JTHNONZEROPOS constraint (having the too long name ITH_POS_DIFFERENT_FROM_0 in [5]) was motivated by a real-life application in molecular biology [8] and can also be used for personnel rostering problems. It has no published propagator, but a decomposition into a conjunction of constraints that have propagators is given in [5]. The NGROUP constraint captures an important combinatorial substructure of the GROUP constraint [5] of the CHIP solver, and has many applications, including personnel rostering.

After a summary of the background material in Section II, the **contributions** and **impact** of Sections III and IV of this paper are as follows:

- From an analysis of checkers corresponding to counter automata, we identify, by means of an automatic invariant generator and other techniques, logically implied constraints that we add to the respective decompositions. In particular, we use the JTHNONZEROPOS and NGROUP constraints as examples.
- We show that the presence of these implied constraints does not increase the time or space complexity of computing the common fixpoint of the propagators of the decomposition. Moreover, we show that these implied constraints often improve propagation, incurring little or no time overhead.
- We show that in the presence of these implied constraints,

GAC can be maintained on the decomposition of the JTHNONZEROPOS constraint, demonstrating that these implied constraints can be quite powerful. However, the objective of our research is to *improve* propagation, regardless of whether or not GAC is maintained on the extended decomposition.

Finally, in Section V, we conclude and discuss related as well as future work.

II. BACKGROUND

To make this paper self-contained, we define the used concepts, namely constraints, constraint problems, solutions, supports, generalised arc consistency (GAC), checker [6], loop invariant [9], disjunctive invariant [10].

A *constraint* is a pair $R(S)$, where $S = \langle w_1, \dots, w_n \rangle$ is a tuple of decision variables, called the *scope*, and R is a set of n-tuples. During search, every decision variable w_i is associated with a current set of possible values, called its *domain* and denoted by $\mathrm{D}(w_i)$. A *solution to a constraint* $R(S)$ is an assignment $\{w_1 = d_1, \dots, w_n = d_n\}$ to its decision variables such that $\langle d_1, \dots, d_n \rangle \in R$. A *constraint problem* is a conjunction of constraints, sometimes given as a set of constraints with an implicit conjunction between its elements. A *solution to a constraint problem* is an assignment to all its decision variables that contains a solution to all its constraints.

The assignment $\{w_k = d_k\}$ is *supported* by a constraint (problem) if there is a solution to that constraint (problem) where $w_k = d_k$ and all decision variables take values in their current domains.

There is *generalised arc consistency* (*GAC*) *on a constraint* $R(S)$ if every domain value of every decision variable of S is supported by $R(S)$; we also say that $R(S)$ is GAC.

There is *GAC on a constraint problem* if every domain value of every decision variable of the problem is supported by the problem; we also say that the problem is GAC.

A *checker* is an algorithm that returns true if and only if an assignment is a solution to a constraint. For example, consider the constraint EXACTLY$(N, \mathcal{V}, P)$, which holds if and only if the sequence $\mathcal{V}$ of decision variables contains exactly N elements taking the given value P. Parameters N and P must be constants, under the restriction $0 \leq N \leq |\mathcal{V}|$. For instance, EXACTLY$(2, [4, 2, 4, 5], 4)$ holds since exactly 2 elements of the sequence $[4, 2, 4, 5]$ take the value 4. A checker for the EXACTLY constraint is given in Algorithm 1.

Informally, a *loop invariant* is a predicate on the variables occurring in the loop, that should be true on entry into a loop and that is guaranteed to remain true on every iteration of the loop. This means that on exit from the loop both the loop invariant and the loop termination condition can be guaranteed. Consider again Algorithm 1. The loop has, among others, the invariant $i \leq |\mathcal{V}|$.

A key problem in automatic invariant generation is the inference of *disjunctive invariants*, which contain at least one disjunction. In order to simplify the generation of disjunctive invariants, we use a technique proposed in [10] to decompose a loop into a semantically equivalent sequence of loops, each of which has only conjunctive invariants. An example will be given in Section III.

Algorithm 1 Checker for the EXACTLY constraint

```
1: function EXACTLY($N$,$\mathcal{V}$,$P$)
2:     $i \leftarrow 0$
3:     $c \leftarrow 0$
4:     while $i < |\mathcal{V}|$ do
5:         if $\mathcal{V}[i] = P$ then
6:             $c \leftarrow c + 1$
7:         $i \leftarrow i + 1$
8:     return $N = c$
```

III. THE JTHNONZEROPOS GLOBAL CONSTRAINT

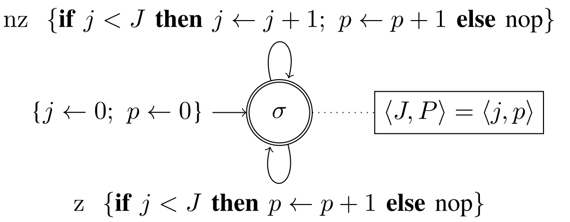
The JTHNONZEROPOS$(J, P, \mathcal{V})$ constraint [5] holds if and only if the J^{th} non-zero element is at position P (counting from 1) of the sequence $\mathcal{V}$ of decision variables. Parameter J must be a constant and parameter P can be a decision variable, under the restriction $1 \leq J \leq P \leq |\mathcal{V}|$. For instance, JTHNONZEROPOS$(2, 4, [5, 0, 0, 1, 3])$ holds since the second non-zero element is at position 4 (namely 1) of $[5, 0, 0, 1, 3]$.

The JTHNONZEROPOS constraint can be used in personnel rostering. For instance, if each element of $\mathcal{V}$ represents the shift of a daily duty of a worker, using the special value 0 for being off-duty, then one can constrain the unknown position P of the J^{th} off-duty day so that it does not occur too early or late.

The constraint was inspired by multiplex dispensation order generation, a real-life problem for DNA sequence analysis with the pyrosequencing method [8]. They used two precursors of this constraint. The FIRST$(\mathcal{V}, P)$ constraint holds if and only if JTHNONZEROPOS$(1, P, \mathcal{V})$ holds. The FOLLOW$([v_1, \dots, v_n], L, Q)$ constraint holds if and only if JTHNONZEROPOS$(1, Q - L, [v_{L+1}, \dots, v_n])$ holds, where parameter L must be a constant and Q can be a decision variable, under the restriction $1 \leq L < Q \leq n$.

A. Automaton-Induced Decomposition

A deterministic finite automaton (taken from [5]) is given in Fig. 1 for JTHNONZEROPOS$(J, P, \mathcal{V})$. The only state, σ, is both the start state (marked by an arrow coming in from no state) and an accepting state (double circle). The alphabet is $\{\mathrm{z}, \mathrm{nz}\}$ rather than the domain of the sequence decision variables v_i of $\mathcal{V}$, because each decision variable v_i is assumed to be paired with a new decision variable s_i, called a *signature decision variable*, which takes the value 'z' (zero) if and only if v_i takes the special value 0, and the value 'nz' (non-zero) otherwise; this can be achieved with $(v_i = 0 \Leftrightarrow s_i = \mathrm{z}) \wedge (v_i \neq 0 \Leftrightarrow s_i = \mathrm{nz})$, called a *signature constraint* (the second conjunct is logically superfluous, but we keep it here for completeness). The automaton is extended [6] with two counters j and p, both initialised to 0; they evolve on each transition as indicated between braces. Until J non-zero elements have been found, counter j maintains the number



nz {**if** $j < J$ **then** $j \leftarrow j+1;\ p \leftarrow p+1$ **else** nop}
$\{j \leftarrow 0;\ p \leftarrow 0\}$
σ
$\langle J, P\rangle = \langle j, p\rangle$
z {**if** $j < J$ **then** $p \leftarrow p+1$ **else** nop}

Fig. 1: Automaton with two counters for the JTHNONZEROPOS$(J, P, \mathcal{V})$ constraint

of *non-zero* values among the elements of $\mathcal{V}$, while counter p maintains the number of *all* values. After J non-zero elements have been found, both counters stop evolving. Consider for example an instance where $J = 2$. After consuming the prefix $[1, 0, 1]$, we have $j_3 = 2$ and $p_3 = 3$. Moreover, for all $i \geq 3$ we have that $j_i = 2$ and $p_i = 3$. Upon acceptance, the final value of the counter pair $\langle j, p\rangle$ is constrained to be equal to $\langle J, P\rangle$; this constraint is called the *acceptance constraint* and is depicted in a box attached to the accepting state.

This counter automaton induces the following decomposition under the framework of [6]:

$$\begin{array}{c} q_0 = \sigma \wedge \langle j_0, p_0\rangle = \langle 0, 0\rangle \wedge \\ \bigwedge_{i=1}^{n} \text{TRANS}(q_{i-1}, \langle j_{i-1}, p_{i-1}\rangle, s_i, q_i, \langle j_i, p_i\rangle) \\ \wedge\ q_n = \sigma \wedge \langle j_n, p_n\rangle = \langle J, P\rangle \\ \wedge\ \bigwedge_{i=1}^{n}(v_i = 0 \Leftrightarrow s_i = \text{z}) \wedge (v_i \neq 0 \Leftrightarrow s_i = \text{nz}) \end{array} \quad (1)$$

The s_i are the already mentioned signature decision variables; notice the signature constraints in the last conjunct of the decomposition. Each q_i is a new decision variable, called a *state decision variable*, denoting the state of the automaton after the signature decision variables $s_1, \ldots, s_i$ have been consumed, with $i \in [0, n]$. Each j_i and p_i is a new decision variable, called a *counter decision variable*, denoting the values of counters j and p of the automaton after the signature decision variables $s_1, \ldots, s_i$ have been consumed, with $i \in [0, n]$. The constraint $\text{TRANS}(q', \langle j', p'\rangle, s, q'', \langle j'', p''\rangle)$ holds if and only if the automaton in Fig. 1 has a transition from state q' to state q'' labelled by symbol s that updates the counters $\langle j, p\rangle$ from values $\langle j', p'\rangle$ to values $\langle j'', p''\rangle$; it is called a *transition constraint*. As this automaton has only one state, one could in principle project the state decision variables away; we keep them to be consistent with the general decomposition of automata in [6].

A folklore result of CP being that GAC is maintained on Berge-acyclic constraint hypergraphs if (but not only if) GAC is maintained on each constraint, in general, in the presence of at least one counter an automaton-induced decomposition has a Berge-cyclic constraint hypergraph, so that GAC *might not* be maintained on such decompositions [6]. In particular, we now show that maintaining GAC on each constraint of the decomposition (1) of the JTHNONZEROPOS constraint *does not* maintain GAC on JTHNONZEROPOS. Consider the ground instance JTHNONZEROPOS$(2, P, [v_1, v_2, 0, v_4])$, with $\text{D}(P) = \{2, \ldots, 4\}$ and unrestricted domains of the v_i. Maintaining GAC on the JTHNONZEROPOS constraint would prune the value 3 from the domain of P, but maintaining GAC on each constraint of the decomposition will not. Indeed, since the element at position 3 is 0, there is no solution where $P = 3$, neither to the decomposition nor to the JTHNONZEROPOS constraint (recall that P is a position where a non-zero element is to occur). To show this, consider the constraints $T_3 = \text{TRANS}(\sigma, \langle j_2, p_2\rangle, s_3, \sigma, \langle j_3, p_3\rangle)$ and $T_4 = \text{TRANS}(\sigma, \langle j_3, p_3\rangle, s_4, \sigma, \langle j_4, p_4\rangle)$ in the decomposition (1), which share the decision variables j_3 and p_3. As illustrated in Fig. 2, there is no solution to T_3 with $\{j_3 = 2, p_3 = 3\}$, even though there are solutions to T_3 where $j_3 = 2$, namely $\{j_2 = 2, p_2 = 2, s_3 = \text{z}, j_3 = 2, p_3 = 2\}$, and to T_3 where $p_3 = 3$, namely $\{j_2 = 1, p_2 = 2, s_3 = \text{z}, j_3 = 1, p_3 = 3\}$. Note that the assignments $\{j_3 = 2, p_3 = 2\}$ and $\{j_3 = 1, p_3 = 3\}$ are contained in solutions to T_4. The assignment $\{j_3 = 2, p_3 = 3\}$ is however also contained in solution $\{j_3 = 2, p_3 = 3, s_4 = \text{z}, j_4 = 2, p_4 = 3\}$ to T_4. In consequence, we have $3 \in \text{D}(p_4)$ because the assignment $\{p_4 = 3\}$ is supported by T_4, even though there is no solution to the decomposition where $P = 3$. Hence, maintaining GAC on the decomposition is not enough to maintain GAC on the JTHNONZEROPOS constraint.

B. Deriving Implied Constraints

The automaton in Fig. 1, together with the signature constraints, can be translated into the checker in Algorithm 2. The automatic invariant generator InvGen [11] derives, in addition to other ones that do not improve propagation (for example $i \leq |\mathcal{V}|$), the following invariants:

$$j \leq J \quad (2)$$

$$0 \leq j \leq p \leq i \quad (3)$$

In order to transform these invariants into implied constraints, we note that, for example, the invariant $p \leq i$ is satisfied at every iteration. On each iteration, the element at position i of the sequence $\mathcal{V}$ is visited. This notion corresponds to the automaton consuming i elements, and so the decision variables j_i and p_i denote the values of the variables j and p after consuming i elements. In consequence, we write the invariants (2) and (3) as the following implied constraints:

$$j_i \leq J \quad (4)$$

$$0 \leq j_i \leq p_i \leq i \quad (5)$$

for $0 \leq i \leq |\mathcal{V}|$. Note that the quantification corresponds to the values of i before and after the loop.

Disjunctive invariants generally arise from the existence of conditionals in the loop body. Given that the disjunctive invariant mode of InvGen is currently experimental, we use the technique proposed by [10] to derive disjunctive implied constraints. Note that not all conditionals imply that a disjunctive invariant exists, but for example, conditionals whose predicate is related to the number of iterations the loop has been executed, as well as predicates that will never be satisfied again after a given numbers of iterations usually do. The idea

$$\begin{tikzcd}
\mathrm{D}(j_0)\times\mathrm{D}(p_0) & \mathrm{D}(j_1)\times\mathrm{D}(p_1) & \mathrm{D}(j_2)\times\mathrm{D}(p_2) & \mathrm{D}(j_3)\times\mathrm{D}(p_3) & \mathrm{D}(j_4)\times\mathrm{D}(p_4) \\
(0,0) \arrow[r, "\mathrm{z}"] \arrow[dr, "\mathrm{nz}"'] & (0,1) \arrow[r, "\mathrm{nz}"] & (1,2) \arrow[r, "\mathrm{z}"] & (1,3) \arrow[r, "\mathrm{nz}"] & (2,4) \\
 & (1,1) \arrow[ur, "\mathrm{z}"] \arrow[r, "\mathrm{nz}"] & (2,2) \arrow[ddr, "\mathrm{z}"] & \boxed{(2,3)} \arrow[r, bend left, "\mathrm{z}"] \arrow[r, bend right, "\mathrm{nz}"'] & \boxed{(2,3)} \\
 & & & \boxed{(1,2)} \arrow[ur, "\mathrm{nz}"] & \\
 & & & (2,2) \arrow[r, bend left, "\mathrm{z}"] \arrow[r, bend right, "\mathrm{nz}"'] & (2,2)
\end{tikzcd}$$

Fig. 2: Solutions to the transition constraints projected onto the induced decision variables of the decomposition (1) of JTHNONZEROPOS$(2, P, [v_1, v_2, 0, v_4])$. Assignments unsupported by (1) are boxed.

is to find a splitter predicate, which informally means that the predicate can be used to divide the loop into a sequence of loops. We split the loop in Algorithm 2 into two loops using the conditional $j < J$ as splitter predicate, obtaining Algorithm 3. Note that both checkers are equivalent. From Algorithm 3, InvGen derives the predicate $j \leq J \wedge p = i$ as an invariant of the first loop, and the predicate $j = J \wedge p \leq i$ as an invariant of the second loop. As a result, the disjunction of both predicates is a disjunctive invariant of the loop in Algorithm 2. These invariants allow us to derive, using standard logic transformations, the following implied constraint:

$$j_i < J \Rightarrow p_i = i \tag{6}$$

We now remove the second loop in Algorithm 3, which does not affect the correctness of the checker, and split the first loop using the splitter predicate $j < J - 1$. After splitting, we simplify the second loop obtaining Algorithm 4. The predicate $\mathcal{V}[i] \neq 0$ is added to the return statement in order to verify that the while loop ended because an element different from 0 was found and not because all the elements in the sequence have been visited. From Algorithm 4, InvGen derives the predicate $j \leq J - 1 \wedge p = i$ as an invariant of the first loop, and the predicate $j = J - 1 \wedge s = \mathrm{z} \wedge p = i$ as an invariant of the second loop. As a result, the disjunction of both predicates is a disjunctive invariant of the loop in Algorithm 2. These invariants allow us to derive, using standard logic transformations, the following implied constraint:

$$(s_i = \mathrm{z} \vee j_{i-1} \neq J - 1) \Rightarrow p_{i+1} \neq i \tag{7}$$

for $0 < i < n$. As can be verified, either by hand or by using a theorem prover (for instance Z3 [12]), if we consider the return condition in Algorithm 4 always to be satisfied, together with the implied constraint (7), we obtain the implied constraint:

$$(s_i = \mathrm{z} \vee j_{i-1} \neq J - 1) \Leftrightarrow p_{i+1} \neq i \tag{8}$$

for $0 < i < n$.

C. The Effect of the Implied Constraints: Maintaining GAC

In the presence of the implied constraints (4), (5), (6), and (8), the transition constraints support only pairs of values for the counter decision variables j_i and p_i that are *reachable* by the counters j and p. In consequence, as proved next, maintaining GAC on each constraint of the extended decomposition will maintain GAC on the decomposition.

Algorithm 2 Checker for the JTHNONZEROPOS constraint

```
function JTHNONZEROPOS($J$,$P$,$\mathcal{V}$)
    $i \leftarrow 0$
    $j \leftarrow 0$
    $p \leftarrow 0$
    while $i < |\mathcal{V}|$ do
        if $j < J$ then
            if $\mathcal{V}[i] \neq 0$ then
                $j \leftarrow j + 1$
            $p \leftarrow p + 1$
        $i \leftarrow i + 1$
    return $j = J \wedge p = P$
```

Algorithm 3 Checker for the JTHNONZEROPOS constraint after splitting the loop once

```
function JTHNONZEROPOS($J$,$P$,$\mathcal{V}$)
    $i \leftarrow 0$; $j \leftarrow 0$
    $p \leftarrow 0$
    while $i < |\mathcal{V}| \wedge j < J$ do
        if $\mathcal{V}[i] \neq 0$ then
            $j \leftarrow j + 1$
        $p \leftarrow p + 1$
        $i \leftarrow i + 1$
    while $i < |\mathcal{V}| \wedge j \geq J$ do
        $i \leftarrow i + 1$
    return $j = J \wedge p = P$
```

Theorem 1. *Assume we add the constraints* (4)*,* (5)*,* (6)*, and* (8)*, for each* $0 < i < n$*, to the decomposition* (1) *of* JTHNONZEROPOS$(J, P, \mathcal{V})$*. If each constraint of the extended decomposition is GAC, then the whole decomposition is GAC.*

Proof: The proof has two parts. First we show that the implied constraints are sound, in the sense that they do not reduce the number of solutions to the (sound decomposition (1) of the) JTHNONZEROPOS constraint. Second we show that

Algorithm 4 Checker for the JTHNONZEROPOS constraint after splitting the loop twice

```
 1: function JTHNONZEROPOS($J$,$P$,$\mathcal{V}$)
 2:     $i \leftarrow 0;\ j \leftarrow 0;\ p \leftarrow 0$
 3:     while $i < |\mathcal{V}| \wedge j < J-1$ do
 4:         if $\mathcal{V}[i] \neq 0$ then
 5:             $j \leftarrow j+1$
 6:         $p \leftarrow p+1$
 7:         $i \leftarrow i+1$
 8:     while $i < |\mathcal{V}| \wedge j \geq J-1 \wedge \mathcal{V}[i] = 0$ do
 9:         $p \leftarrow p+1$
10:         $i \leftarrow i+1$
11:     return $j = J-1 \wedge p = P-1 \wedge \mathcal{V}[i] \neq 0$
```

it is always possible to construct a solution to the constraint using every value in every domain after GAC is maintained on each constraint in the extended decomposition.

Soundness. It is easy to show that the constraints (4) to (6) are sound. The implied constraints (8) capture that if a given signature decision variable s_i is assigned the value 'z', then there is no solution to the JTHNONZEROPOS constraint where $P = i$. Moreover, if there are fewer than $J-1$ signature decision variables that can take the value 'nz' among s_1 till s_{i-1}, then there is no solution to the JTHNONZEROPOS constraint were $P = i$.

Constructing solutions. For each value in the domain of each decision variable, we will show that if the value is supported by the extended decomposition, then it is part of a solution to the constraint. Recall that parameter J is a constant, satisfying $1 \leq J \leq P \leq n = |\mathcal{V}|$. In this decomposition there are five classes of decision variables: the sequence decision variables v_i; the state decision variables q_i; the counter decision variables j_i and p_i; the signature decision variables s_i; and the problem decision variable P. Each sequence decision variable v_i appears in only *one* constraint (a signature constraint), with scope $\langle v_i, s_i \rangle$, so that the v_i need not be considered because maintaining GAC on the signature constraints will be enough, since we can show that every domain value of every signature decision variable is supported by the decomposition. Also, we do not consider the state decision variables q_i because each is necessarily set to σ, because σ is the only state.

The s_i decision variables. For any given s_i we will show *every* value in $\mathrm{D}(s_i)$ is supported by the constraint, by showing there exists some value k in $\mathrm{D}(P)$ and values in the domains of the other signature decision variables giving a solution to the constraint. We will not consider all the decision variables p_i and j_i as their values will be fixed by transition constraints once values have been assigned to the s_i and P.

For a chosen s_i, we will show that if there is some k in $\mathrm{D}(P)$ with $k < i$, then the signature decision variables can be divided into two groups: the decision variables $s_{k+1}, \ldots, s_i, \ldots, s_n$, which can be assigned any value in their domains and be part of an assignment that satisfies the constraint; and the decision variables $s_1, \ldots, s_k$, where there must be exactly J decision variables that can be assigned the 'nz' value and exactly $k - J$ decision variables taking the 'z' value to give a solution when P is assigned k.

We will now show that such an assignment is possible, when k is less than i. If we assign to P the value k, then no values are pruned from the domain of the chosen s_i. By (8) we know that s_k has the domain $\{\mathrm{nz}\}$; further, by (8) and by the TRANS constraints, the decision variable j_k will have the domain $\{J\}$. By the transition constraints and the implied constraints (5) and (6) for each $0 \leq i \leq k$ the decision variable p_i will have the domain $\{i\}$, and again by the transition constraints $\mathrm{D}(p_{k+1}) = \mathrm{D}(p_{k+2}) = \cdots = \mathrm{D}(p_n) = \{k\}$, because, after assigning P to be k, the decision variable j_k will be equal to J, and j_0 is always assigned the value 0 in the decomposition. When the j_i decision variables are assigned, by the transition constraints each j_{i+1} has the value j_i or the value j_i+1. Hence when GAC is maintained on the transition constraints, it is guaranteed that there are exactly J signature decision variables taking the 'nz' value and exactly $k - J$ signature decision variables taking the 'z' value in the sequence $s_1, \ldots, s_k$, because the domains of the j_i and p_i decision variables are constrained by the values in the domains of the s_i decision variables.

For a chosen s_i, if there is no k in $\mathrm{D}(P)$ with $k < i$, then we have to show that, after assigning s_i any value in its domain, the domain of P will never be empty. If $\mathrm{D}(P)$ has exactly one element $k \geq i$, then we know that the decision variables $s_{k+1}, \ldots, s_n$ can be assigned any values for satisfying the constraint. Then, as in the previous paragraph when we assigned P the value k, j_k will be assigned J, and hence when GAC is maintained on the transition constraints it is guaranteed that there are exactly J signature decision variables taking the value 'nz' and exactly $k-J$ signature decision variables taking the value 'z' in the sequence $s_1, \ldots, s_k$. If the domain of $\mathrm{D}(P)$ has more than one element, then assigning s_i any value in its domain will remove at most one element from $\mathrm{D}(P)$. Hence, for an assignment of s_i, any value in $\mathrm{D}(P)$ can be picked to give a solution extending the assignment of s_i as in the case where we assumed that $\mathrm{D}(P) = \{k\}$.

The P decision variable. After assigning P any value k in its domain, by the implied constraints (8) and after propagation the decision variable s_k will be assigned 'nz' and the decision variable j_{k-1} will be assigned $J-1$. If 'nz' is not in $\mathrm{D}(s_i)$ or $J-1$ is not in $\mathrm{D}(j_{k-1})$, then by the implied constraints (8) the value k would not be in $\mathrm{D}(P)$. Again the signature decision variables can be divided into two groups: the decision variables $s_{k+1}, \ldots, s_n$ can be assigned any values in their domains in order to satisfy the constraint; and the decision variables $s_1, \ldots, s_k$, where there must be exactly J decision variables taking the 'nz' value and exactly $k - J$ decision variables taking the 'z' value. Since P is assigned k, after propagation p_k will be assigned k and j_k is assigned J. As before, because $j_0 = 0$ and $j_k = J$ when GAC is maintained on the transition constraints it will be guaranteed that there are exactly J signature decision variables taking the 'nz' value and exactly $k-J$ signature decision variables taking the 'z' value in the sequence $s_1, \ldots, s_k$.

The decision variables p_i and j_i. After GAC is maintained on all constraints in the extended decomposition, the domains of the p_i and j_i are constrained by the TRANS constraints and the values in the domains of the s_i decision variables. Hence, for each value in the domain of a p_i or j_i it is possible using the arguments above to pick values for the s_i and P to satisfy the constraint. ■

GAC can be maintained on each implied constraint individually, since J and i are constants for each of them, and on the transition constraints [6].

As a sanity check, we implemented in SICStus Prolog version 4.2.1 [13] the extended decomposition of JTHNONZEROPOS$(J, P, \mathcal{V})$. We generated instances with random amounts ($n \leq 50$) of signature decision variables $[s_1, \ldots, s_n]$ as well as random initial domains of P (one value, two values, and intervals of length 2 or 3) and the s_i (one value, and binary domains). The extended decomposition is never faster than the original decomposition (but 20% slower on average); such a time overhead is in practice probably compensated by fewer invocations of the propagators of the other constraints of the problem. Note that we did not use the built-in automaton decomposition in SICStus Prolog in our experiments. The reason is that, in general, its transition constraint does not maintain GAC. If we use our own GAC implementation (outlined in the following sub-section) of the TRANS constraint, then this sanity check can also be used to search for counterexamples to GAC on the decomposition: upon many millions of generated random instances, no such counterexample was found, lending further credence to Theorem 1.

D. Complexity of GAC on the Extended Decomposition

Maintaining GAC on the JTHNONZEROPOS$(J, P, \mathcal{V})$ constraint decomposition (1) takes $\Omega(n^2)$ time and $\Omega(n)$ space. In order to obtain this lower bound, one would need to maintain GAC on each transition constraint in constant time and use constant space to store the domain of each induced decision variable. Using the multi-valued decision diagram constraint [14], we managed to maintain GAC on each transition constraint in $O(n)$ time, giving $O(n^3)$ time and $O(n^3)$ space on the whole decomposition: we omit the details, as they are not needed for proving that, using the lower bound, the added implied constraints bear no asymptotic overhead on maintaining GAC on the decomposition.

Theorem 2. *Maintaining GAC on the implied constraints* (4)*,* (5)*,* (6)*, and* (8)*, does not increase the asymptotic time and space complexity of maintaining GAC on the original decomposition* (1) *of the* JTHNONZEROPOS$(J, P, \mathcal{V})$ *constraint.*

Proof: There are $4(n-1)$ implied constraints. Each requires constant time and is at worst woken $|\mathrm{D}(P)|$ times down any branch of the search tree, that is $O(n)$ times. Propagating all implied constraints down a branch of the search tree therefore takes $O(n^2)$ time. Hence, the time complexity of the decomposition is not affected even if GAC is maintained on the transition constraints in constant time.

Since none of the implied constraints requires a new data structure or extra decision variables, there is no asymptotic extra space required. Hence, the space complexity of the decomposition is not affected by the implied constraints. ■

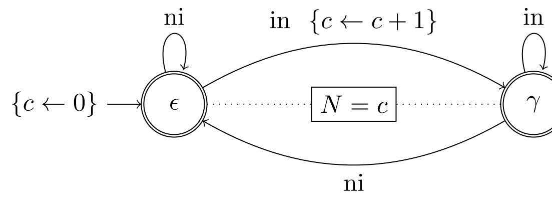


Fig. 3: Predicate counter automaton (with one counter) for the NGROUP$(N, \mathcal{V}, W)$ constraint.

IV. THE NGROUP CONSTRAINT

In a sequence, a *group* is a contiguous subsequence with values from a given set. Here we consider only part of the GROUP constraint [5]. The NGROUP$(N, \mathcal{V}, W)$ constraint holds if and only if there are N groups of values from the set W in the sequence $\mathcal{V}$ of decision variables. For example, NGROUP$(3, [2,4,1,6,4,3,4], \{2,4,6\})$ holds since there are three groups of even values in $[2,4,1,6,4,3,4]$, namely $[2,4]$, $[6,4]$, and $[4]$.

A. Automaton-Induced Decomposition

A counter automaton (taken from [5]) is given in Fig. 3 for NGROUP$(N, \mathcal{V}, W)$, using the notation and terminology of Section III. The start state is called ϵ and is reached if the most recently consumed symbol (if any) is not in a group. The other state is called γ and is reached if the most recently consumed symbol is in a group. Both states are accepting. The alphabet is $\{\mathrm{in}, \mathrm{ni}\}$, because the signature constraint $(v_i \in W \Leftrightarrow s_i = \mathrm{in}) \wedge (v_i \notin W \Leftrightarrow s_i = \mathrm{ni})$ pairs the sequence decision variable v_i with a new signature decision variable s_i whose domain is that alphabet. Counter c maintains the number of groups within the prefix of $\mathcal{V}$ consumed so far. The acceptance constraint is $N = c$ for both accepting states. This automaton, together with the signature constraints, can be translated into the checker given in Algorithm 5.

This counter automaton induces the following constraint decomposition:

$$\begin{gathered} q_0 = \epsilon \wedge c_0 = 0 \\ \wedge\ \mathrm{TRANS}(q_0, c_0, s_1, q_1, c_1) \wedge\ \cdots \\ \wedge\ \mathrm{TRANS}(q_{n-1}, c_{n-1}, s_n, q_n, c_n) \\ \wedge\ c_n = N\ \wedge \\ \textstyle\bigwedge_{i=1}^{n} (v_i \in W \Leftrightarrow s_i = \mathrm{in}) \wedge (v_i \notin W \Leftrightarrow s_i = \mathrm{ni}) \end{gathered} \tag{9}$$

where the new decision variables are the signature decision variables s_i, the state decision variables q_i, and the counter decision variables c_i. The transition constraint TRANS(q, c', s, q', c'') holds if and only if the automaton in Fig. 3 has a transition from state q to state q' labelled by symbol s that updates the counter c from value c' to value c''.

We now consider the constraint instance $\text{NGROUP}(N, [v_1, v_2, v_3, v_4], \{2, 4, 6\})$. The signature constraints are $s_i = 1 \Leftrightarrow v_i \in \{2, 4, 6\}$ and $s_i = 0 \Leftrightarrow v_i \notin \{2, 4, 6\}$. The value of N is set by propagation to the value of the induced counter decision variable c_4, and hence the domain of N is the set $\{0, \ldots, 4\}$. GAC on all the constraints in the decomposition does not suffice to tighten the upper bound of the domain of each decision variable c_i to $\lfloor i/2 \rfloor$.

B. Deriving Implied Constraints

The automaton in Fig. 3, together with the signature constraints, can be translated into the checker in Algorithm 5. Note that in the automaton in Fig. 3, every path of two transitions increases the counter value by at most 1. Let us extend the checker in Algorithm 5 in order to keep track of the previous values of the counter variables c, obtaining the checker in Algorithm 6. Variable c_1 denotes the value of variable c at the previous iteration, and variable c_2 denotes its value two iterations ago. From the checker in Algorithm 6, InvGen derives the invariants:

$$c_2 \leq c \leq c_2 + 1 \tag{10}$$

Note that c_1 does not appear on the invariants, and we use it only to keep track of the previous value of c. We translate the invariants (10) into the implied constraints:

$$c_{i-1} \leq c_{i+1} \leq c_{i-1} + 1 \tag{11}$$

for $0 < i < n$.

At this point we believe that the implied constraints $(q_i = \epsilon \wedge s_j = \text{ni}) \Rightarrow c_i < c_j$ can also be derived from loop invariants and would suffice to maintain GAC on the NGROUP constraint decomposition. However, the long-term objective of our research is to automatically extend automaton-induced decompositions by adding implied constraints that often strictly improve propagation, without significantly slowing the propagation down, if slowing it down at all. Whether or not these implied constraints actually help maintain GAC on the decomposition is out of the scope of our objective.

C. The Effect of the Implied Constraints

Even though after adding the implied constraints (11) propagation is improved, GAC on every constraint in the extended decomposition is not enough to maintain GAC on the NGROUP constraint. For example, consider again the constraint instance $\text{NGROUP}(N, [v_1, v_2, v_3, v_4], \{2, 4, 6\})$. After the assignment $s_2 = \text{in}$, there is at least one group in the sequence $\mathcal{V}$, that is $N > 0$, but GAC on every constraint is not enough to prune the value 0 from the domains of all c_i with $i \geq 2$, and so 0 is not pruned from the domain of N.

Towards testing the implied constraints, we implemented in SICStus Prolog version 4.2.1 [13] the original and extended decompositions of $\text{NGROUP}(N, \mathcal{V}, W)$. We generated instances with random amounts ($n \leq 50$) of signature decision variables $[s_1, \ldots, s_n]$ as well as random initial domains of N (one value, two values, and intervals of length 2 or 3) and the

Algorithm 5 Checker for the NGROUP constraint
```
1: function NGROUP($N$,$\mathcal{V}$,$W$)
2:     $c \leftarrow 0$
3:     $q \leftarrow \epsilon$
4:     $i \leftarrow 0$
5:     while $i < |\mathcal{V}|$ do
6:         if $\mathcal{V}[i] \in W$ then
7:             if $q = \epsilon$ then
8:                 $c \leftarrow c + 1$
9:                 $q \leftarrow \gamma$
10:        else
11:            $q \leftarrow \epsilon$
12:        $i \leftarrow i + 1$
13:    return $N = c$
```

Algorithm 6 Checker for the NGROUP constraint keeping track of previous counter values
```
1: function NGROUP($N$,$\mathcal{V}$,$W$)
2:     $c \leftarrow 0;\ c_1 \leftarrow 0;\ c_2 \leftarrow 0$
3:     $q \leftarrow \epsilon$
4:     $i \leftarrow 0$
5:     while $i < |\mathcal{V}|$ do
6:         $c_2 \leftarrow c_1$
7:         $c_1 \leftarrow c$
8:         if $\mathcal{V}[i] \in W$ then
9:             if $q = \epsilon$ then
10:                $c \leftarrow c + 1$
11:                $q \leftarrow \gamma$
12:        else
13:            $q \leftarrow \epsilon$
14:        $i \leftarrow i + 1$
15:    return $N = c$
```

s_i (one value and binary domains). Note that, in the presence of the signature constraints, generating random domains for the signature decision variables is equivalent to generating random domains for the decision variables $[v_1, \ldots, v_n]$. Upon many millions of such instances, it turns out that the extended decomposition is never slower than the original decomposition (but 2% faster on average), but always prunes at least as many values (but 105% more on average) and detects at least as many failures (but 8% more on average) as the latter.

D. Complexity of the Extended Decomposition

Maintaining GAC on the implied constraints (11) does not increase the asymptotic time and space complexity of maintaining GAC on the original decomposition (9) of the $\text{NGROUP}(N, \mathcal{V}, W)$ constraint. The proof is similar to the one of Theorem 2 in Section III-D, based on the observation that there are $2(n-1)$ implied constraints.

V. Conclusion, Related Work, and Future Work

Counter automata provide a uniform representation format for many constraints. We believe that automatically deriving implied constraints that are necessary for maintaining GAC, or that simply improve propagation, on counter-automaton-induced constraint decompositions can be seen as an automated way to design propagators. Proving manually that a candidate (extended) decomposition maintains GAC is very tedious: witness the long and hard proof for the JTHNONZEROPOS constraint in Theorem 1. It took us a very long time to find this proof, partly because we initially manually derived the implied constraints, starting from an analysis of the failure to prove GAC. Manual proofs are also error-prone, which is why we also ran the sanity check on the extended decomposition of JTHNONZEROPOS at the end of Section III-C.

The hypergraphs of the decompositions (1) and (9) are actually α-acyclic [15]. The main objective of our research is to examine when and how (a consistency level close to) GAC is maintained on Berge-cyclic (for instance α-acyclic) constraint hypergraphs (rather than a deep desire to settle open questions about the little known constraints studied in this paper, very useful though they are).

A. Related Work

It was observed [6] that an α-acyclic constraint hypergraph maintains GAC when all constraints are pairwise consistent [16], but no further analysis was given.

There is also a large body of related work (e.g., [17], [18], [19], [20]) on decomposing global constraints *manually* in order to maintain GAC on the whole decomposition. This paper can be seen as a *more systematic* approach to maintaining GAC, via automaton-induced decompositions.

Initial experiments with using automated reasoning systems towards inferring implied algebraic constraints from a constraint problem are reported in [21], [22], both using an extension [21] of the PRESS equation solver. In contrast, the present work uses a loop invariant generator; it is aimed at a specialised class of constraint problems (namely counter-automaton-induced decompositions) and at a specialised class of implied constraints (on the induced counter decision variables), and is therefore more successful.

There is also some related work using graph invariants to systematically derive implied constraints in [23] in order to improve efficiency. Our paper explores a different approach, which is capable of finding other invariants and it does not require a database of invariants.

B. Future Work

We have given a methodology to infer useful implied constraints, using an automated invariant generator, that can be added to an automaton-induced decomposition. We will now automate as many steps as possible of this methodology, for the AUTOMATON constraint in SICStus Prolog.

ACKNOWLEDGEMENTS

The authors are supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council (VR). Many thanks to the anonymous referees for their helpful comments.

REFERENCES

[1] A. Aggoun and N. Beldiceanu, "Extending CHIP in order to solve complex scheduling and placement problems," *Mathematical and Computer Modelling*, vol. 17, no. 7, pp. 57–73, 1993.

[2] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[3] M. Carlsson, N. Beldiceanu, and J. Martin, "A geometric constraint over k-dimensional objects and shapes subject to business rules," in *CP 2008*, ser. LNCS, P. J. Stuckey, Ed., vol. 5202. Springer, 2008, pp. 220–234.

[4] S. Bourdais, P. Galinier, and G. Pesant, "HIBISCUS: A constraint programming application to staff scheduling in health care," in *CP 2003*, ser. LNCS, F. Rossi, Ed., vol. 2833. Springer, 2003, pp. 153–167.

[5] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit, "Global constraint catalogue: Past, present, and future," *Constraints*, vol. 12, no. 1, pp. 21–62, March 2007, the current working version of the catalogue is at http://www.emn.fr/z-info/sdemasse/aux/doc/catalog.pdf.

[6] N. Beldiceanu, M. Carlsson, and T. Petit, "Deriving filtering algorithms from constraint checkers," in *CP 2004*, ser. LNCS, M. Wallace, Ed., vol. 3258. Springer, 2004, pp. 107–122.

[7] G. Pesant, "A regular language membership constraint for finite sequences of variables," in *CP 2004*, ser. LNCS, M. Wallace, Ed., vol. 3258. Springer, 2004, pp. 482–495.

[8] M. Carlsson and N. Beldiceanu, "Multiplex dispensation order generation for pyrosequencing," in *Proceedings of the Workshop on CSP Techniques with Immediate Application (held at CP 2004)*, 2004.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[10] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *CAV 2011*, ser. LNCS, vol. 6806. Springer, 2011, pp. 703–719.

[11] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *CAV 2009*, ser. LNCS, vol. 5643. Springer, 2009, pp. 634–640.

[12] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS 2008*, ser. LNCS. Springer, 2008, pp. 337–340.

[13] M. Carlsson, G. Ottosson, and B. Carlson, "An open-ended finite domain constraint solver," in *PLILP 1997*, ser. LNCS, vol. 1292. Springer, 1997, pp. 191–206.

[14] K. C. Cheng and R. H. Yap, "An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints," *Constraints*, vol. 15, no. 2, pp. 265–304, Apr. 2010.

[15] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, "On the desirability of acyclic database schemes," *Journal of the ACM*, vol. 30, no. 3, pp. 479–513, July 1983.

[16] R. Fagin, "Degrees of acyclicity for hypergraphs and relational database schemes," *Journal of the ACM*, vol. 30, no. 3, pp. 514–550, July 1983.

[17] C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh, "Circuit complexity and decompositions of global constraints," in *IJCAI 2009*, C. Boutilier, Ed., 2009, pp. 412–418.

[18] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, C.-G. Quimper, and T. Walsh, "Reformulating global constraints: The *slide* and *regular* constraints," in *SARA 2007*, ser. LNAI, vol. 4612. Springer, 2007, pp. 80–92.

[19] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh, "Decomposition of the NValue constraint," in *CP 2010*, ser. LNCS, D. Cohen, Ed., vol. 6308. Springer, 2010, pp. 114–128.

[20] C.-G. Quimper and T. Walsh, "Decomposing global grammar constraints," in *CP 2007*, ser. LNCS, C. Bessière, Ed., vol. 4741. Springer, 2007, pp. 590–604.

[21] B. Hnich, J. Richardson, and P. Flener, "Towards automatic generation and evaluation of implied constraints," Department of Information Technology, Uppsala University, Sweden, Tech. Rep. 2003-014, originally written in August 2000, available at www.it.uu.se/research/reports/2003-014.

[22] A. M. Frisch, I. Miguel, and T. Walsh, "Extensions to proof planning for generating implied constraints," in *Proceedings of Calculemus 2001*, 2001, pp. 130–141, available at http://www.calculemus.net/meetings/siena01/.

[23] N. Beldiceanu, M. Carlsson, J.-X. Rampon, and C. Truchet, "Graph invariants as necessary conditions for global constraints," in *CP 2005*, ser. LNCS, P. van Beek, Ed., vol. 3709. Springer, 2005, pp. 92–106.