Semi-Automatic Generation of CHR Solvers from Global Constraint Automata

Frank Raiser

Faculty of Engineering and Computer Sciences, University of Ulm, Germany
Frank.Raiser@uni-ulm.de

Abstract. Constraint programming often involves global constraints, which have been cataloged in [1] and for which various custom filtering algorithms have been published. This work presents a semi-automatic generation of CHR solvers for the set of global constraints which can be defined by specific automata described in [2]. The solvers only need to be generated once and achieve arc-consistency for over 40 global constraints.

1 Introduction

Global constraints are a combination of multiple constraints used to perform additional filtering on the domains of involved variables. While it is common knowledge, that specialized filtering algorithms on global constraints can often achieve better domain pruning than generic approaches, the development of such algorithms requires a lot of effort. Thus, in [2] a generic method for deriving filtering algorithms from special checker automata is introduced.

Constraint Handling Rules (CHR) is a multi-headed, guarded, and concurrent constraint programming language. CHR was designed for writing constraint solvers and is increasingly being used as a general-purpose programming language. However, there is no direct support for global constraints available in CHR so far. The aim of this work is to adapt the results from [2] in order to generate CHR solvers for global constraints. To this end we make use of the Prim-Miner algorithm proposed in [3] to generate rules from a constraint logic program (CLP). The necessary CLP is automatically created from the description of the automata corresponding to a global constraint given by the global constraint catalog (GCC) [1].

This work presents a refined version of the Prim-Miner algorithm that adapts it to the problem at hand, which results in a significant improvement of runtime complexity. It further presents the generation of the required CLP and analyses the properties of the set of resulting CHR rules. Using this approach we can semi-automatically generate CHR solvers for over 100 global constraints, including arc-consistent solvers for about 40 of them.

The following section provides necessary preliminaries on CHR, the checker automata proposed in [2], and the Prim-Miner algorithm from [3]. Section 3 discusses all steps of the semi-automatic generation of CHR solvers, including post-processing and experimental results.

2 Preliminaries

In the following we introduce the necessary notions of automata for global constraints [2], Constraint Handling Rules [4], and the generic Prim-Miner algorithm [3]. We assume basic familiarity of the reader with constraints and constraint logic programming.

2.1 Automata for Global Constraints

In [2] automata for checking if a global constraint holds are introduced. The underlying idea is to compile a list of *signature arguments* from the arguments of the global constraint and use this list to iterate through the automaton. To each signature argument a signature constraint is associated and a transition is made in the automaton if the corresponding signature constraint holds. Despite the creation of signature arguments, the other difference to traditional finite automata are counters: a counter is initialized to a value in the start state of the automaton and can be modified at each transition. Additionally, the final state specifies a final value which has to hold for the counter in order for the global constraint to hold.

Example 1. Figure 1 shows the automaton for the `among` $(N, [X_1, \ldots, X_k], \overline{V})$ constraint. This constraint holds if exactly N variables from the set of variables $X_1, \ldots, X_k$ take a value in the set $\overline{V}$. For the automaton to check if the constraint holds it tests whether $X_i \in \overline{V}$. If this is the case the counter c is incremented, otherwise it remains unchanged. After processing all X_i the final value of the counter has to equal N.

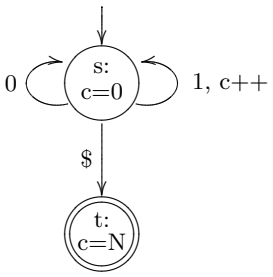

Fig. 1. Automaton for the `among` constraint

More formally an automaton is defined in [2] as
$\langle \mathit{Signature}, \mathit{SignatureDomain}, \mathit{SignatureArg}, \mathit{Counters}, \mathit{States}, \mathit{Transitions}, s, t \rangle$
where:

- $\mathit{Signature} = [S_0, \ldots, S_{m-1}]$ is the list of signature variables

- *SignatureDomain* specifies the domain for the S_i variables ($0 \leq i < m$)
- *SignatureArg* = $[\Delta_0, \ldots, \Delta_{m-1}]$ are signature arguments, which are associated to the *Signature* by *signature constraints* $\psi(S_i, \Delta_i)$ for $0 \leq i < m$. Each Δ_i is a subset of variables of the corresponding global constraint, and $|\Delta_i| = k$ for a constant k and all $i = 0, \ldots, m-1$.
- *Counters* is a list of counters. Each counter is given as a term t(*Counter*, *InitialValue*, *FinalVariable*) with *Counter* being a symbolic name, *InitialValue* an initial integer value, and *FinalVariable* a variable which is unified with the counter value in the final state of the automaton.
- *States* is the set of states of the automaton.
- *Transitions* is the set of transitions given by either $\text{arc}(id_1, l, id_2)$, for arcs without counter modifications, or $\text{arc}(id_1, l, id_2, \overline{c})$ for arcs including counter modifications. id_1 and id_2 correspond to the source and target states, l is the value the signature variable S_i should have, and $\overline{c}$ gives the values of counters after the transition. Usual arithmetic functions such as $+, -, \min$, or $\max$ can be used.
- s and t are the starting and final state, respectively.

Example 2. The automaton for the `among` $(N, [X_1, \ldots, X_k], \overline{V})$ constraint can be formally defined as $\langle [S_1, \ldots, S_k], \{0, 1\}, [X_1, \ldots, X_k], [t(c, 0, N)], \{s, t\}$, $\{\text{arc}(s, 0, s), \text{arc}(s, 1, s, [c \leftarrow c + 1])\}, s, t\rangle$.

In this case the $\psi(S_i, X_i)$ constraint is specified by $(S_i = 0 \leftrightarrow X_i \notin \overline{V}) \wedge (S_i = 1 \leftrightarrow X_i \in \overline{V})$.

We introduce the following notations for the remainder of this work:

- Let $\mathcal{M} = \{1, \ldots, |\mathcal{M}|\}$ be the set of all possible values for signature variables. Note that $\mathcal{M}$ can be split up into a union of disjoint sets where each set corresponds to the possible values of signature variables for outgoing transitions of a single state.
- Each value $i \in \mathcal{M}$ corresponds to a transition, which in turn, corresponds to a signature constraint denoted as C_i.
- Let $D(\Delta_i)$ denote the set of domains of all variables in Δ_i.
- Let the domains for the signature variables S_i be $D(S_i) \subseteq \mathcal{M}$ for $i = 0, \ldots, m-1$.
- Let $C(S_i) = \{C_j \mid j \in D(S_i)\}$
- We write $\Delta \notin C$, iff for all values for the variables in Δ taken from $D(\Delta)$ the resulting tuple $x \notin C$.

In [2] it is further explained how to derive a filtering algorithm from these automata. It is based on arc-consistent solvers for the ψ constraints as well as ϕ constraints, which are used to encode the transitions. A pair of constraints $\psi(S_i, \Delta_i) \wedge \phi(Q_i, \overline{K_i}, S_i, Q_{i+1}, \overline{K_{i+1}})$ is added for every signature argument, with Q_i and Q_{i+1} being variables representing the current and next state, $\overline{K_i}$ and $\overline{K_{i+1}}$ being lists of variables for the counters before and after the transition, and S_i being the signature variable. Furthermore, instead of Q_0 the starting state is

used and for S_m only a $\phi(Q_m, \overline{K_m}, \$, t, \overline{K_{m+1}})$ constraint is added for a unique signature value \$ and the final state t.

The proposed generic filtering algorithm is arc-consistent if the solvers for the ψ and ϕ constraint are arc-consistent. In this work we generate CHR solvers for those constraints, which in combination with rules generating the necessary ψ and ϕ constraints allow arc-consistent filtering for global constraints. Note however, that the arc-consistency result only holds for automata in which all subsets of variables in *SignatureArg* are pairwise disjoint and which do not involve counters. In all other cases the filtering algorithm and therefore the generated CHR solvers can still be used, but may not achieve arc-consistent filtering.

It is important to point out, that the automata proposed in [2] can have $O(n)$ states with n being the number of signatures. As our approach is directed to the generation of solvers which can be reused for all instances of a specific global constraint it is only applicable to those constraints for which the automaton has a constant size.

Example 3. For the `among` $(N, [X_1, \ldots, X_k], \overline{V})$ constraint the following conjunction of ψ and ϕ constraints is used:

$$\begin{array}{ll} \psi_{\overline{V}}(S_1, X_1) & \wedge\phi(s, [0], S_1, Q_2, \overline{K_2})\wedge \\ \psi_{\overline{V}}(S_2, X_2) & \wedge\phi(Q_2, \overline{K_2}, S_2, Q_3, \overline{K_3})\wedge \\ \ldots & \ldots \\ \psi_{\overline{V}}(S_k, X_k) & \wedge\phi(Q_k, \overline{K_k}, S_k, Q_{k+1}, \overline{K_{k+1}})\wedge \\ & \wedge\phi(Q_{k+1}, \overline{K_{k+1}}, \$, t, [N]) \end{array}$$

In this example $\psi_{\overline{V}}$ denotes, that the specification of the ψ constraint refers to $\overline{V}$. It is shown later, that for the realization of ψ constraints in CHR further arguments are needed depending on the signature constraints.

2.2 Constraint Handling Rules

This section presents the syntax and operational semantics of Constraint Handling Rules [4]. Constraints are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are provided by the constraint solver while user-defined constraints are defined by a CHR program. For our purpose we only need a subset of CHR, namely *propagation rules*. Propagation rules are of the form

$$\mathit{Rulename} \ @ \ H_1, \ldots, H_i \Rightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

where *Rulename* is a unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, the guard $\overline{G} = G_1, \ldots, G_j$ is a conjunction of built-in constraints and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints.

The operational semantics of a propagation rule is based on an underlying constraint theory CT for the built-in constraints and a state, which is a pair $\langle G, C\rangle$ where G is a goal, i.e. a conjunction of user-defined and built-in constraints, and C is a (built-in) constraint. A propagation rule of the form

$H \Rightarrow \overline{G} \mid B$ is applicable to a state $\langle E \wedge G, C\rangle$ if $CT \models \forall(C \rightarrow \exists\overline{x}((H \doteq E) \wedge \overline{G}))$ where $\overline{x}$ are the variables in H. We define the following state transition for the application: $\langle E \wedge G, C\rangle \mapsto \langle E \wedge G \wedge B, (H \doteq E) \wedge C \wedge \overline{G}\rangle$.

Another type of rule available in CHR are simplification rules, which are of the form

$$Rulename \ @ \ H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

Their operational semantics is similar to propagation rules, except that simplification rules remove the constraints matched to $H_1, \ldots, H_i$ from the goal. Applicability is the same as for propagation rules and the state transition for a simplification rule is defined as $\langle E \wedge G, C\rangle \mapsto \langle G \wedge B, (H \doteq E) \wedge C \wedge \overline{G}$.

2.3 Prim-Miner Algorithm

The Prim-Miner algorithm [3] automatically generates CHR solvers for a given logical specification of a constraint. In this work we apply it to generate a solver for the ϕ constraint.

The basic idea of the algorithm is to use an underlying constraint logic program (CLP) P which specifies a constraint $Base_{LHS}$. Then sets of candidates for the left-hand and right-hand sides of CHR rules are tested against this CLP. For every subset C_{LHS} of the left-hand side candidates set it is tested, if $Base_{LHS} \cup C_{LHS}$ is consistent. It the test fails with respect to P a failure rule $(Base_{LHS} \cup C_{LHS} \Rightarrow \bot)$ is created. Otherwise all candidates d from the set of right-hand side candidates are tested in the goal $Base_{LHS} \cup C_{LHS} \cup \{not(d)\}$ against the CLP P. If this goal fails d must be a logical consequence of $Base_{LHS} \cup C_{LHS}$ and is included in the right-hand side of the generated CHR propagation rule with head $Base_{LHS} \cup C_{LHS}$.

As the Prim-Miner algorithm presented in [3] is very generic we slightly modify it using additional knowledge available in the setting of this work. Section 3.3 presents the application of the algorithm to our problem, details on the modifications made, and their performance implications.

3 Semi-Automatic Solver Generation

To generate a CHR solver for a global constraint the following steps are required:

1. extract the automaton definition from the global constraint catalog [1]
2. write CHR rules for creating signature arguments as well as ψ and ϕ constraints
3. write/generate solver for ψ constraints
4. automatically generate solver for ϕ constraints
5. optionally post-process the created ruleset

The generation of the CHR solver is semi-automatic, because steps 1–3 require some manual labor. Section 3.1 describes a generic way to generate ψ and

```
1  C(Δ, X, K_f) ⇔ C'(Δ, X, s, K_s, K_f).
2  C'([], X, Q, K, K_f) ⇔ φ(Q, K, $, t, K_f).
3  C'([Δ | Δ'], X, Q, K, K_f) ⇔
4          S ∈ D̂_S ∧ Q' ∈ D̂_Q ∧ ψ(S, Δ, X)∧
5          φ(Q, K, S, Q', K') ∧ C'(Δ', X, Q', K', K_f).
```

Listing 1.1. Creation of ψ and ϕ constraints

ϕ constraints in CHR for the case that the global constraint is already given by its signature arguments. Section 3.2 presents the generation of the solver for the ψ constraints and Section 3.3 the application of the Prim-Miner algorithm to generate the solver for the ϕ constraints. Section 3.4 details the post-processing step which reduces the number of rules in the final solver. The complexity of the resulting solver is discussed in Section 3.5, before Section 3.6 presents experimental results.

3.1 Generation of ψ and ϕ Constraints

For the filtering algorithm proposed in [2] the generation of ψ and ϕ constraints for the automaton is required. Note that this step cannot be fully automated, as the signature arguments depend on the specific global constraint for which to generate a solver. However, assuming that the signature arguments are already made available as $\overline{\Delta}$, we can canonically create the required constraints. Let $C(\overline{\Delta}, \overline{X}, \overline{K_f})$ be the CHR representation of the global constraint with $\overline{X}$ being a sequence of arguments used for determining if a ψ constraint holds and $\overline{K_f}$ being the final values of counters.

Example 4. The `among` $(N, [X_1, \ldots, X_k], \overline{V})$ global constraint involves a fixed set $\overline{V}$ of values and the signature constraint is given by $S_i \leftrightarrow (X_i \in \overline{V})$. Therefore, the among constraint is given in CHR as `among` $([X_1, \ldots, X_k], [\overline{V}], [N])$.

Using the above representation of the global constraint in CHR and the information contained in the automaton retrieved from the global constraint catalog [1], Listing 1.1 presents a generic way of creating the ψ and ϕ constraints. In line 1 the generation of the constraints is forwarded to a special C' constraint, thereby already enforcing the starting state s and the starting values $\overline{K_s}$ for the counters. The C' constraint is then used for a recursive generation of the ψ and ϕ constraints as shown in lines 3–4, before the final ϕ constraint is generated in line 2 including the \$ transition into the final state t with the final values $\overline{K_f}$ for the counters. In Listing 1.1 $\hat{D}_Q$ denotes the set of all states of the automaton and $\hat{D}_S$ denotes the set of all transitions, i.e. $\hat{D}_S = \mathcal{M}$. For the `among` constraint it holds that $\hat{D}_S = \{0, 1\}$ and $\hat{D}_Q = \{s, t\}$.

3.2 Generation of Solver for ψ Constraint

The generation of a solver for the ψ constraints assumes, that all signature constraints C_i and their negations are available as built-in constraints. We also require that union of subsets of these constraints are available as built-in constraints. This directly leads to the creation of rules of the following kind $\forall i \in \mathcal{M}$:

$$\psi(S, \Delta) \Rightarrow \Delta \notin C_i \mid S \in \mathcal{M} \setminus \{i\}.$$

Intuitively these rules make use of each transition i corresponding to a signature constraint C_i and state the fact that if this constraint does not hold the transition cannot be made. Thus, the corresponding identifier for the transition is removed from $D(S)$.

As is shown later, these rules already suffice for arc-consistency of $D(S)$. In order to achieve arc-consistency on the domains in $D(\Delta)$, however, further rules are required. Considering a domain restriction for $D(S) = \{i_1, \ldots, i_k\}$ with $i_1, \ldots, i_{|\mathcal{M}|}$ being a permutation of $\mathcal{M}$ and $0 < k \leq |\mathcal{M}|$, a constraint $C_{i_1} \cup \ldots \cup C_{i_k}$ can be propagated. As stated earlier, we require all of these constraints, as well as their union, to be available as arc-consistent built-in constraints.

$$\psi(S, \Delta) \wedge S \in \{i_1, \ldots, i_k\} \Rightarrow \Delta \in (C_{i_1} \cup \ldots \cup C_{i_k})$$

Such rules are inserted for $D(S) = \{i_1, \ldots, i_k\}$ being any of the possible subsets of $\mathcal{M}$ with the exception of $\emptyset$. In the case of $D(S) = \emptyset$ the global constraint is already shown to be inconsistent. Note that the case $D(S) = \mathcal{M}$ can yield domain restrictions for $D(\Delta)$ if $D(\Delta) \setminus (C_1 \cup \ldots \cup C_{\mathcal{M}}) \neq \emptyset$.

Demanding the availability of the union of these constraints as built-in constraints is justified by the following observation: it is always possible to use the built-in solvers of the single signature constraints C_i to construct a brute-force solver for their union by iterating through all values x in the domain Δ and successively testing them against each C_j of the union. As is discussed in Section 3.5, the runtime complexity of such a solver is suboptimal and a custom solver for each union of signature constraints is preferable.

The seemingly obvious addition of rules based on positive built-in constraints as guards, like $\psi(S, \Delta) \Rightarrow \Delta \in C_i \mid S \in \{i\}$, is in fact incorrect. For two constraints C_i and C_j with $C_i \cap C_j \neq \emptyset$ corresponding to outgoing transitions of two different states, the domain $D(S) = \{i, j\}$ must not be reduced to $\{i\}$ solely because $\Delta \in C_i$. It may also hold that $\Delta \in C_j$, as only constraints corresponding to outgoing transitions of the same state are required to be mutually incompatible.

Assuming an automaton with $|\mathcal{M}|$ transition edges where each edge is labeled with a different constraint, $O(|\mathcal{M}|)$ rules of the first kind and $O(2^{|\mathcal{M}|})$ rules of the second kind are added. Using these $O(2^{|\mathcal{M}|})$ rules we have the following result:

Theorem 1. *The generated solver achieves arc-consistency for the ψ constraint.*

The fully automatic generation of these rules is hindered, because the built-in constraints C_i and their negations are difficult to extract from the automaton's

```
1  ψ(S, Δ, V) ⇒ Δ ∈ V | S ∈ {0, 1} \ {0}
2  ψ(S, Δ, V) ⇒ Δ ∉ V | S ∈ {0, 1} \ {1}
3
4  ψ(S, Δ, V) ∧ S ∈ {0} ⇒ Δ ∉ V
5  ψ(S, Δ, V) ∧ S ∈ {1} ⇒ Δ ∈ V
6  ψ(S, Δ, V) ∧ S ∈ {0, 1} ⇒ ⊤
```

Listing 1.2. Solver for ψ constraint of `among` constraint

definition. Furthermore, the S and Δ arguments may be insufficient to specify the C_i constraints, such that additional information is required which is only available from an instance of the global constraint. In the previous example of an `among` constraint the necessary built-in constraint is set inclusion, however, the specific set $\overline{V}$ is given by the global constraint's instance. Therefore, generation of the solver for the ψ constraint is can only be partly automated.

Example 5. The generated solver for the ψ constraint of the `among` global constraint is shown in Listing 1.2. As mentioned earlier, the solver needs knowledge about the specific set $\overline{V}$ which, therefore, has to be passed to the ψ constraint as an additional argument. The first two rules filter the domain of S, while the latter three rules filter the domain of Δ. As the two transitions constraints C_1 and C_2 are set inclusion and exclusion, every value of Δ has to satisfy at least one of those constraints. Thus, the last rule – which assumes in the guard, that either transition is possible – cannot propagate any additional information, due to $D(\Delta) \setminus (C_1 \cup C_2) = \emptyset$.

3.3 Generation of Solver for ϕ Constraint

The solver for the ϕ constraint is based on the Prim-Miner algorithm. The basic idea is to encode the automaton in a constraint logic program P for the algorithm to work with and use all possible domains as candidate inputs. For $\phi(Q, \overline{K}, S, Q', \overline{K'})$ varying domains for Q, Q', and S are added and given to the CLP P which checks if a solution is possible for these domains.

In the following, we discuss the automatic generation of the CLP P, the application of the Prim-Miner algorithm, and the modifications made to the original Prim-Miner algorithm in order to improve its runtime complexity when applied to this specific problem.

Generation of CLP Creating the CLP P for the ϕ constraints is a straightforward encoding of the automaton's transitions into CLP rules. Listing 1.3 shows how transitions of the kind $\text{arc}(q, s, q', [c = f(c), \ldots])$ for the `among` constraint are encoded as rules. The generation of these CLPs can be fully automated given the definition of the automaton.

```
1  φ(Q, [K], S, Q', [K']) :- Q = s ∧ Q' = s ∧ S = 0 ∧ K' = K .
2  φ(Q, [K], S, Q', [K']) :- Q = s ∧ Q' = s ∧ S = 1 ∧ K' = K + 1 .
3  φ(Q, [K], S, Q', [K']) :- Q = s ∧ Q' = t ∧ S = $ ∧ K' = K .
```

Listing 1.3. CLP for `among` constraint

A check performed by the Prim-Miner algorithm against such a CLP consists of a default backtracking search. If the given domain restrictions are consistent with one of the automaton's transitions the check succeeds, and the check fails if the given domain restrictions do not allow for any of the transitions to fire. In fact a successful check means the CLP P has found a support σ for the ϕ constraint according to the following definition:

Definition 1 (Support). *Given a constraint C and domains $D_1, \ldots, D_n$ a tuple $\sigma \in D_1 \times \ldots \times D_n$ is called a* support *of $D_1, \ldots, D_n$ for C if $\sigma \in C$.*

Solver Generation The generation of the solver for the ϕ constraint is performed by the GC-Prim-Miner algorithm shown in Listing 1.4, which is a modified version of the Prim-Miner algorithm. It uses the previously generated CLP P against which to test goals. As mentioned before, the input candidate domains $L_Q, L_{Q'}, L_S$ are the power sets of the sets of all states and transitions, respectively, except for the empty set. The resulting ruleset is a CHR solver for the ϕ constraint providing arc-consistency for global constraints whose automaton is free of counters:

Theorem 2. *For automata which do not involve counters the resulting rule set achieves arc-consistency for ϕ.*

The original Prim-Miner algorithm only allows one set of candidates as input and then iterates over all its subsets. As the candidate domains are already power sets the input to the Prim-Miner algorithm is of size $O(2^n + 2^{|\mathcal{M}|})$ for an automaton with n states and $|\mathcal{M}|$ transitions. Iterating over all subsets of this input then yields a runtime of $O(2^{2^n} \cdot 2^{2^{|\mathcal{M}|}})$ for the Prim-Miner algorithm's main loop which needs to be improved in order to be feasible.

However, the main part of the algorithm in lines 10–27 remains almost unchanged. First a candidate left-hand side for a possible CHR rule is selected. The selection process differs from the original Prim-Miner algorithm and is detailed later. Given the selected candidate C_{LHS} the goal $Base_{LHS} \cup C_{LHS}$ is tested against the CLP P, where $Base_{LHS}$ consists of a single ϕ constraint. Therefore, this test ensures that the chosen left-hand side is consistent, i.e. there are supports for the ϕ constraint with the given domain restrictions in C_{LHS}.

In the original Prim-Miner algorithm [3] a special failure rule is created in those cases the goal $Base_{LHS} \cup C_{LHS}$ fails. As Lemma 1 shows such a rule has no effect on the propagation power of the generated ruleset. Thus, the GC-Prim-Miner algorithm omits the generation of failure rules.

```
1  R = ∅
2  L is a list of all subsets of {Q, Q', S} (L ≠ ∅)
3  L_Q, L_Q', L_S are lists of candidate domains for variables Q, Q', S
4   ordered by size of the domains
5  while L is not empty do
6   Remove from L its first element denoted C_Vars
7   Let L_x be the Cartesian product of all candidate domains
8    for all variables in C_Vars
9   while L_x is not empty do
10   Remove from L_x its first element denoted C_LHS.
11   if the goal (Base_LHS ∪ C_LHS) succeeds
12    with respect to the CLP P then
13     let rhs = ∅.
14     for all v ∈ {Q, Q', S} do
15      for all d ∈ L_v do
16       if the goal (BaseLHS ∪ C_LHS ∪ {not(d)}) fails
17        with respect to the CLP P then
18         add d to the set rhs.
19         break % break inner for, continue with next variable v
20       endif
21      endfor
22     endfor
23     if rhs ≠ ∅ then
24      add the rule (Base_LHS ∪ C_LHS ⇒ rhs) to R
25     endif
26   endif
27  endwhile
28 endwhile
29 output R
```

Listing 1.4. GC-Prim-Miner Algorithm

Lemma 1 (no failure rules). *Any generated failure rules – i.e. rules with body* $\bot$ *– are redundant.*

In the inner loop in lines 13–22 candidates for the right-hand side are determined. Again the selection of the candidate d is modified. The goal $Base_{LHS} \cup C_{LHS} \cup \{not(d)\}$ is tested against P for failure. If this goal fails there is no support anymore when the additional domain restriction $not(d)$ is added. Therefore, all supports must take values from the domain given in d and d can be added to the right-hand side of the CHR rule. In lines 23–25 the CHR rule is added to the resulting ruleset if a successful right-hand side domain restriction was found.

Example 6. Applying the GC-Prim-Miner algorithm to the CLP created from the automaton of the `among` constraint generates, among others, the following rules:
$\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{\$, 0\} \wedge Q' \in \{t\} \Rightarrow Q \in \{s\} \wedge S \in \{\$\}$
$\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge Q \in \{s\} \wedge S \in \{\$, 0\} \wedge Q' \in \{t\} \Rightarrow S \in \{\$\}$
As the first rule's head is more general and as it propagates everything the second rule could propagate it is sufficient, so that the second rule can be removed due to redundancy. This process is explained in more detail in Section 3.4.

As the runtime complexity of the direct application of the Prim-Miner algorithm is insufficient we can make use of the specifics of our application to improve it. Given a list of all subsets of possible values for all variables and then considering all subsets of this list will often result in goals like $Q \in \{0, 1\} \wedge Q \in \{0\}$, which supply multiple domain restrictions for the same variable. Considering, that the intersection of these domains is again a domain restriction which is part of the original input all of these goals can be ignored. The same argument holds for the right-hand side candidates.

Therefore, if we order the domain restrictions for a variable by the size of their domains we can simply iterate through them. For the outer loop this means we first loop over all subsets of the variables and then in lines 3–5 over the different combinations of domain restrictions for each of the variables. Similarly in lines 14–15 the inner loop iterates over all variables and their domain restrictions. Breaking out of the loop in line 19 is possible due to the size ordering of the domain restrictions, as it ensures that the best possible domain restriction for this variable was found. Note that it is irrelevant, that the ordering is incomplete: if two domain restrictions of equal size could be used, only the values in the intersection of the domains belong to supports, as otherwise the tests in line 16 could not have failed. However, the intersection of these domains occurs earlier in the ordered list and was therefore already causing a break of the loop.

This tailored version of the Prim-Miner algorithm performs much better: The outer-most loop iterates over all non-empty subsets of variables. As in our case we only have the three variables Q, Q', and S, this means 7 iterations. The loop in lines 9–27 iterates over the Cartesian product of all candidate domains. In the worst case this happens for all three variables resulting in $2^n * 2^n * 2^{|\mathcal{M}|}$ loop executions. The inner loop in line 14 simply iterates over the three variables, before the loop in lines 15–21 is run for each domain restriction of a variable, i.e.

at most $2^n + 2^n + 2^{|\mathcal{M}|}$ times. Therefore, the overall complexity of the GC-Prim-Miner algorithm – more precisely the number of calls to the underlying CLP – is $O(7{*}2^n{*}2^n{*}2^{|\mathcal{M}|}{*}3{*}(2^n{+}2^n{+}2^{|\mathcal{M}|})) = O(2^{2n+|\mathcal{M}|}{*}(2^n{+}2^{|\mathcal{M}|})) = O(2^{3n+|\mathcal{M}|} + 2^{2n+2|\mathcal{M}|})$ which is a major improvement over the original $O(2^{2^n} * 2^{2^{|\mathcal{M}|}})$.

3.4 Post-Processing of Rule Set

After generating a set of CHR rules with the GC-Prim-Miner algorithm the resulting rule set can be reduced. As discussed earlier, the unmodified Prim-Miner algorithm would create rules with multiple domain restrictions for a variable. The GC-Prim-Miner algorithm already eliminates these rules, but nevertheless, a large number of generated rules is redundant in a similar way to the ones presented in Example 6. Therefore, an additional post-processing of the rule set leads to a more concise solver.

In order to find redundant rules for removal, the results about operational equivalence of CHR programs in [5] can be applied. [5] presents a decidable, sufficient, and necessary syntactic condition to determine operational equivalence of CHR programs that are terminating and confluent. We can apply this condition by removing each rule from the generated rule set successively, and check if the complete rule set and the rule set without that rule are operationally equivalent. If they are, the selected rule is redundant and can be removed.

To be able to apply this result, terminating and confluent CHR programs are required. Confluence is explained in detail in [6], so that we only refer to the intuitive notion of confluence here: A CHR program is confluent if for every choice of rules to apply there is always a sequence of rule applications resulting in the same state. In case of a terminating program this notion is even stronger: it states, that every terminating computation beginning with an initial goal, i.e. a computation resulting in a state to which no more rules are applicable, results in the same final state, no matter in which order the rules are applied.

Confluence of our generated global constraint solvers is easy to see: for the simplification rules in Listing 1.1 only one of them is applicable at any time, so that all terminating computations need to fully unroll the global constraint into ψ, ϕ, and domain constraints. The solvers for the ψ and ϕ constraints in turn are solely based on propagation rules. As these rules only add new domain restrictions to achieve arc-consistency, the order of their application is irrelevant. Therefore, the generated solvers are confluent.

As for termination, it is important to note that a propagation rule could be applied an infinite number of times leading to a trivial non-termination. As this behavior is not wanted, a token store was introduced in [6], which remembers on which set of constraints a propagation rule was applied, such that the rule cannot be applied on the same set of constraints again. Using this restriction on the application of propagation rules, termination of the generated solvers is guaranteed: the unrolling into ψ and ϕ constraints is terminating due to the finite number of arguments to the global constraint. The applications of the propagation rules terminate due to the token store.

1 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \Rightarrow Q \in \{s\}$
2 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge Q' \in \{t\} \Rightarrow Q \in \{s\} \wedge S \in \{\$\}$
3 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge Q' \in \{s\} \Rightarrow Q \in \{s\} \wedge S \in \{0, 1\}$
4 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{\$\} \Rightarrow Q \in \{s\} \wedge Q' \in \{t\}$
5 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{0\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$
6 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{1\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$
7 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{0, 1\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$

Listing 1.5. Solver for ϕ constraint for the `among` automaton

Example 7. Using the GC-Prim-Miner algorithm on the CLP for the ϕ constraint for the `among` automaton and removing all redundant rules generates the solver given in Listing 1.5.

3.5 Runtime Complexity

The runtime complexity of solvers for global constraints based on automata is discussed in [2], so we only discuss the complexity of the three parts specific to this work: the generation of the required constraints and the solvers for ψ and ϕ constraints.

The generation of the ψ and ϕ constraints given in Listing 1.1 is a simple linear recursion. The solver for ϕ constraints is strong enough to achieve arc-consistency through the application of a single rule. As the GC-Prim-Miner algorithm creates a rule for every possible starting condition, that propagates the maximal amount of information, a second rule application is unnecessary.

Therefore, the runtime complexity depends on the solver for the ψ constraint. As the generated rules require the signature constraints of the automaton to be checked in the guard the runtime complexity is based on the runtime used for these entailment checks. The first kind of rules generated for the ψ constraint solver can be applied at most $O(|\mathcal{M}|)$ times, as each time a value from $\mathcal{M}$ is removed from $D(S)$. Note again, that the use of a token store to avoid trivial non-termination guarantees, that a rule is not applied multiple times for the same ψ constraint.

The second kind of rule is applied once to propagate the maximal union of signature constraints available for $D(S)$. As a single such rule application achieves arc-consistency on $D(\Delta)$, the runtime complexity depends on the complexity of the solver for the union of these built-in constraints. Using the above-mentioned brute-force approach the complexity is bound by $O(D(\Delta) \cdot b)$, where b is the complexity of the slowest–to–solve built-in constraint. It is thus important to provide custom built-in solvers for the union of signature constraints, especially as b is often constant.

3.6 Experimental Results

We have applied our approach to several global constraints and Table 3.6 shows the number of rules of the resulting solvers for a selection of global constraints. The number of rules for the given ϕ solver are the rules produced by the GC-Prim-Miner algorithm, including redundant rules. Our experiments have shown, that approximately two thirds of such rules are redundant.

Table 3.6 shows, that the total number of rules is clearly dominated by the number of rules required for the ϕ constraint solver. Furthermore, it provides confirmation for the number of rules being exponential to the number of states and signature constraints of the automaton. It also shows, that the number of rules for the ψ constraint solver only depends on the number of signatures, whereas the number of rules for the ϕ constraint solver depends on the actual automaton's transitions.

The last column gives an indication of the running time required for the solver generation by listing the number of calls the GC-Prim-Miner algorithm makes to its underlying CLP program. Note that the actual number of calls is significantly lower, than the asymptotic bounds given in Section 3.3. This is due to the test in lines 11–12 in Listing 1.4 which often causes the inner two loops to be avoided.

4 Conclusion

In this paper we have shown a way to semi-automatically generate CHR solvers for the set of automata-describable global constraints. The process is not fully automated due to the generation of signature arguments and because signature constraints are not available in a suitable format in the global constraint catalog. Nevertheless, our approach applies to over 40 different global constraints for which CHR solvers can be generated.

We have shown that by the use of the GC-Prim-Miner algorithm and given the availability of built-in constraint solvers for signature constraints the generated CHR solvers achieve arc-consistency in those cases the automata-based filtering proposed in [2] allows for it. We have further shown, that the generality of the Prim-Miner algorithm can cause a runtime complexity problem, which can be alleviated by an order of magnitude if specialized for the problem at hand.

Our experimental results have shown, that the exponential number of rules generated reduces the approach to be used for small automata only. However, the automata published in the global constraint catalog so far, consist of very few states and transitions only, except for automata with a linear number of states to which our approach is not applicable at all.

For future work the problems associated with the ψ constraint solver need to be tackled. As there are few semantically different signature constraints used in the various automata it might be possible to develop arc-consistent solvers for these, including their negations and unions. Together with a way to automatically extract the signature constraints from the definitions given in the global

Table 1. Comparison of number of generated rules

Name	States	Signatures	ψ rules	ϕ rules	CLP calls
arith	2	2	2	11	99
arith_or	2	2	2	11	99
decreasing	2	2	2	11	99
domain_constraint	2	2	2	11	99
elem	2	2	2	11	99
element_greatereq	2	2	2	11	99
in_same_partition	2	2	2	11	99
increasing	2	2	2	11	99
lex_different	2	2	2	11	99
not_all_equal	2	2	2	11	99
stage_element	2	2	2	11	99
int_value_precede	2	3	5	27	287
sequence_folding	2	3	5	27	295
maximum	3	3	5	185	2195
minimum	3	3	5	185	2195
two_orth_are_in_contact	3	3	5	178	2023
element_sparse	3	4	10	423	5257
no_peak	3	4	10	439	6175
no_valley	3	4	10	439	6175
minimum_greater_than	3	6	36	1851	63959
between_min_max	4	3	5	1002	14763
global_contiguity	4	3	5	996	15241
minimum_except_0	4	5	19	4599	105157
and	5	3	5	4538	88959
nand	5	3	5	4526	89159
nor	5	3	5	4526	86471
or	5	3	5	4538	86367
equivalent	6	2	2	8829	229007
imply	6	2	2	8125	192339
xor	6	2	2	8829	229007

constraint catalog this would allow for a fully automated generation of the CHR solvers.

References

1. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalog: Past, present and future. Constraints **12**(1) (2007) 21–62
2. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In Wallace, M., ed.: Principles and Practice of Constraint Programming (CP'2004). Volume 3258 of Lecture Notes in Computer Science., Springer-Verlag (2004) 107–122
3. Abdennadher, S., Rigotti, C.: Automatic generation of CHR constraint solvers. TPLP **5**(4-5) (2005) 403–418
4. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (October 1998) 95–138
5. Abdennadher, S., Frühwirth, T.: Operational equivalence of chr programs and constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming (CP 1999). Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57
6. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Principles and Practice of Constraint Programming. (1997) 252–266

Appendix (Proofs)

Lemma 1 (no failure rules). *Any generated failure rules – i.e. rules with body* $\bot$ *– are redundant.*

Proof. *Let* $r \equiv H \Rightarrow \bot \equiv \phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge X_1 \in D_1 \wedge \ldots \wedge X_k \in D_k \Rightarrow \bot$ *be a failure rule with* $X_i \in \{Q, Q', S\}, 1 \leq i \leq k \leq 3$*, and* $X_i \neq X_j$ *for* $i \neq j$*. W.l.o.g. there is no failure rule* r' *with a head* $H' \subset H$*, i.e. a head with fewer domain restrictions, otherwise* r *is redundant due to* r'*. Let* $H_i = H \setminus \{X_i \in D_i\}$ *for some* i*. Then there is no rule* $H_i \Rightarrow \bot$*, and thus, there exists a support in the domain restrictions given in* H_i *(otherwise the corresponding test in the GC-Prim-Miner algorithm would generate such a rule). Therefore, for* $\tilde{D}_i = \hat{D}_i \setminus D_i$ *with* $\hat{D}_i$ *being the initial maximal domain of* X_i*, the GC-Prim-Miner algorithm tests the goal* $\langle H_i \wedge \neg(X_i \in \tilde{D}_i)\rangle \equiv \langle H_i \wedge X_i \in D_i\rangle$*. As this goal fails, the rule* $r' \equiv H_i \Rightarrow X_i \in \tilde{D}_i$ *is generated. Now consider a CHR goal* $\langle \tilde{H}, C\rangle$ *to which rule* r *is applicable, i.e.* $X_i \in D_i$ *is contained in* $\tilde{H}$*. As* $H_i \subset H$*, rule* r' *is also applicable to this goal, thus* $\langle \tilde{H}, C\rangle \overset{r'}{\mapsto} \langle \tilde{H} \wedge X_i \in \tilde{D}_i, (\tilde{H} \doteq H_i) \wedge C\rangle \equiv \langle \bot\rangle$*, because* $X_i \in D_i \wedge X_i \in \tilde{D}_i$ *is inconsistent.* □

Theorem 1. *The generated solver achieves arc-consistency for the* ψ *constraint.*

Proof. *We have to distinguish two cases for* $\psi(S, \Delta)$*: arc-consistency on* $D(S)$ *and arc-consistency on* $D(\Delta)$*:*

Let $i \in D(S)$ *be a value for which the given* ψ *constraint has no support. This means there is no accompanying variable assignment* $x \in D(\Delta)$ *satisfying* $x \in C_i$*.*

Therefore, the constraint theory $CT \models \Delta \notin C_i$*, which causes the generated rule* $\psi(S, \Delta) \Rightarrow \Delta \notin C_i \mid S \in \mathcal{M} \setminus \{i\}$ *to be applied. The solver for the domain constraints then propagates that* $S \in D(S) \cap \mathcal{M} \setminus \{i\} = D(S) \setminus \{i\}$*, thus removing the value* i *from the domain of* S *to achieve arc-consistency.*

On the other hand let $i \in D(S)$ *with support* σ*. Then* $\exists x \in D(\Delta)$ *satisfying* $x \in C_i$*, therefore,* $CT \not\models \Delta \notin C_i$*. So, the corresponding rule to remove* i *from* $D(S)$ *is not applicable. As the other kind of rules only affect* $D(\Delta)$*, the value* i *is not removed from* $D(S)$ *by any of the generated rules.*

Let now $x \in D(\Delta)$ *be an assignment of the variables in* Δ *for which there is no support. Then* x *is no support for the constraints* $C_i \forall i \in D(S)$*. As* $D(S) = \{i_1, \ldots, i_k\}$ *is w.l.o.g. non-empty, the rule* $\psi(S, \Delta) \wedge S \in \{i_1, \ldots, i_k\} \Rightarrow \Delta \in (C_{i_1} \cup \ldots \cup C_{i_k})$ *can be applied to propagate the corresponding built-in constraint. As* $x \in D(\Delta)$*, but* $x \notin (C_{i_1} \cup \ldots \cup C_{i_k})$*, this results in* x *being removed from* $D(\Delta)$*.*

Let $x \in D(\Delta)$ *with support* σ*. Then* $\exists i \in D(S)$ *such that* $x \in C_i$*. Then it holds for all rules of the kind* $\psi(S, \Delta) \wedge S \in D(S) \Rightarrow \Delta \in (C_{i_1} \cup \ldots \cup C_{i_k})$*, that* $i \in \{i_1, \ldots, i_k\}$*, and therefore,* $x \in (C_{i_1} \cup \ldots \cup C_{i_k})$*. Thus, if such a rule is applied and arc-consistency is enforced on the union of these built-in constraints, the value* x *cannot be removed from* $D(\Delta)$*. Analogously to the above case, the rules responsible for removing values from* $D(S)$ *do not affect* $D(\Delta)$ *and a supported value* x *can therefore not be removed from* $D(\Delta)$*.* □

Theorem 2. *For automata which do not involve counters the resulting rule set achieves arc-consistency for* ϕ*.*

Proof. Let $x \in D(Q)$*, for which the given* ϕ *constraint has no support. Let* C_{LHS} *be the conjunction of domain constraints including* $Q \in D(Q)$*, which is tested during the GC-Prim-Miner algorithm. If the goal* $Base_{LHS} \cup C_{LHS}$ *fails, a failure rule would be generated. However, according to Lemma 1 such a rule is redundant, i.e. the inconsistency occurs due to other rules.*
Otherwise the GC-Prim-Miner algorithm considered $Q \in D(Q)$ *for the left-hand side. As* $Q \in D(Q) \setminus \{x\}$ *is among the candidates for the right-hand side – in fact all domains smaller than* $D(Q) \setminus \{x\}$ *are tested first, but would only result in an even stronger propagation – the goal* $Base_{LHS} \cup C_{LHS} \cup \{not(Q \in D(Q) \setminus \{x\})\}$ *is tested for failure during the GC-Prim-Miner algorithm. As* $Q \in D(Q) \wedge \neg(Q \in D(Q) \setminus \{x\})$ *implies* $Q = x$*, calling the CLP with this goal is equivalent to finding a support for* $Q = x$ *for the* ϕ *constraint, which by assumption does not exist. Therefore the goal fails and the GC-Prim-Miner algorithm generates the rule for* $Base_{LHS} \cup C_{LHS} \Rightarrow Q \in D(Q) \setminus \{x\} \wedge \ldots$*, possibly including some more candidates for the right-hand side. Applying this rule removes the value* x *from domain* $D(Q)$*, thus achieving arc-consistency.*

Unsupported values are removed analogously from the domains of Q' *and* S*. Therefore, it remains to be shown, that no rule application can result in the removal of a supported value:*

Let $x \in D(Q)$ *with support* σ*. Assume a generated rule with* $Q \in D(Q)$ *on the left-hand side is applicable and the right-hand side contains the constraint* $Q \in D'$*. As the rule was created by the GC-Prim-Miner algorithm, it holds*

that the CLP call for $C_{Base} \cup C_{LHS} \cup \{not(Q \in D')\}$ *failed. However,* $Q \in D(Q) \wedge \neg(Q \in D')$ *implies* $Q \in D(Q) \setminus D'$*. If* $x \notin D'$ *the test would not have failed due to* σ*, thus,* $x \in D'$ *and the rule application cannot remove* x *from* $D(Q)$*. Again, the argumentation is analogous for the domains of* Q' *and* S*.* □