Propagator Groups

Mikael Z. Lagerkvist and Christian Schulte

KTH – Royal Institute of Technology, Sweden
{zayenz,cschulte}@kth.se

Abstract. This paper introduces propagator groups as an abstraction for controlling the execution of propagators as implementations of constraints. Propagator groups enable users of a constraint programming system to program how propagators within a group are executed.

The paper exemplifies propagator groups for controlling both *propagation order* and *propagator interaction*. Controlling propagation order is applied to debugging constraint propagation and optimal constraint propagation for Berge-acyclic propagator graphs. Controlling propagator interaction by encapsulating failure and entailment is applied to general reification and constructive disjunction. The paper describes an implementation of propagator groups (based on Gecode) that is applicable to any propagator-centered constraint programming system. Experiments show that groups incur little to no overhead and that the applications of groups are practically usable and efficient.

1 Introduction

Over the last two decades, an array of techniques to control the execution of groups of propagators have been developed:

- S-boxes [10] support the debugging of constraint propagation by grouping several propagators into a conceptually bigger propagator that typically captures some problem structure. Then, the drastically fewer executions of the bigger propagator can be understood by a modeler during debugging. While it is generally acknowledged that the lack of support for debugging is one of the key obstacles to learning constraint programming, none of today's mainstream systems support it.
- The optimal propagation of Berge-acyclic propagator graphs requires to order the execution of a group of propagators and is generally acknowledged to be significant for efficient propagation [16, 3]. However, no mainstream constraint programming system implements it.
- General reification (reifying arbitrary constraints and groups of constraints) and constructive disjunction are well known concepts that can improve propagation considerably [13]. Both require to control the execution and interaction of groups of propagators. However, both concepts are only supported to some extent by Propia [15] and general reification by Mozart [18, 19].

I.P. Gent (Ed.): CP 2009, LNCS 5732, pp. 524–538, 2009.

In summary, these powerful and even essential techniques are known but either not widely available or not even implemented. All these techniques have in common that they are based on controlling the execution of groups of propagators. This paper introduces *propagator groups* as a new abstraction that supports the user-level implementation of the above mentioned techniques. A group collects a set of propagators and defers their scheduling and execution to a user-supplied routine. This makes scheduling and execution programmable, while the implementation of propagator groups requires only very small and local changes to the underlying constraint programming system.

Similar to groups, spaces in Mozart [19] also encapsulate failure and entailment. However, the lack of control over propagation order and access to local variables makes them unable to implement propagator ordering and constructive disjunction. The Propia system [15] supports constructive disjunction and to some extent generalized reification through a nested search procedure. However, it can not express propagator ordering.

Contributions. This paper contributes *propagator groups* as a new abstraction for constraint programming systems that allows many interesting techniques to be implemented by a user without modifying the underlying system. The paper contributes a simple yet expressive architecture for groups and techniques for their efficient implementation. Moreover, the paper shows how propagator groups can be applied to debugging, optimal propagator ordering, general reification, and constructive disjunction. These applications are shown to be efficient with propagator groups. In particular, the paper contributes the first implementation of optimal propagation for Berge-acyclic propagator graphs.

Plan of the paper. In the following section the necessary background on constraint programming is given. In Sect. 3 a model and implementation of propagator groups is presented. Sections 4 and 5 apply groups to debugging, optimal propagation ordering, general reification, and constructive disjunction. The final section concludes.

2 Constraint Programming

Constraint programming is based on two important concepts, variables (together with their associated domains) and constraints.

Variables and domains. There is a finite set of variables Var and a finite set of values Val. A *domain* $d \in Dom$ is a set of values a variable can take, $Dom = \mathcal{P}(Val)$. A *store* $s \in Store$ is a complete mapping from variables to domains, $Store = Var \to Dom$. An *assignment* is a store where the range of the function is restricted to singleton sets ($\{\{v\} \mid v \in Val\}$).

Set relations $\sim$ are lifted to a pair of stores S_1 and S_2 in the natural, pointwise way ($S_1 \sim S_2 = \forall x \in Var.\, S_1(x) \sim S_2(x)$). A store S_1 is *stronger* than a store S_2, written $S_1 \leq S_2$, if $S_1 \subseteq S_2$. A store S_1 is *strictly stronger* than a store

S_2, written $S_1 < S_2$, if $S_1 \leq S_2$ and $S_1 \neq S_2$. The *disagreement set* $\mathrm{dis}\,(S_1, S_2)$ of stores S_1 and S_2 is defined as $\{x \in \mathit{Var} \mid S_1(x) \neq S_2(x)\}$.

A tuple of values over variables $x_1, \ldots, x_n$ can be turned into a store in the following way: for a variable $x \in \mathit{Var}$, $\mathrm{store}\,(\langle v_1, \ldots, v_n\rangle)\,(x)$ is defined as $\{v_l\}$ if $x = x_l$ for some $l \in \{1, \ldots, n\}$ and as Val otherwise.

Constraints. A *constraint* $c \in \mathit{Con}$ over the set of variables $\mathrm{var}\,(c) = \{x_1, \ldots, x_n\}$ is defined by the set of assignments that are solutions to the constraint, $\mathit{Con} = \mathcal{P}(\{\langle v_1, \ldots, v_n\rangle \mid v_i \in \mathit{Val}\})$. A store can be turned into a constraint over variables $x = \{x_1, \ldots, x_n\}$ using $\mathrm{cons}\,(S, x) = \{\langle v_1, \ldots, v_n\rangle \mid \forall i. v_i \in S(x_i)\}$.

A *constraint satisfaction problem* (CSP) is a pair of a set of constraints C and a store S, $\langle C, S\rangle \in \mathcal{P}(\mathit{Con}) \times \mathit{Store}$. A tuple $\langle v_1, \ldots, v_n\rangle$ in a constraint over variables $x_1, \ldots, x_n$ is *valid* under the store S iff $\forall i. v_i \in S(x_i)$. An assignment a is a *solution* to the CSP $\langle C, S\rangle$, both over variables $\{x_1, \ldots, x_n\}$, iff the assignment is a solution for each constraint, $\forall c \in C. \langle a(x_1), \ldots, a(x_n)\rangle \in c$, and the assignment is valid for the store, $a \subseteq S$. The solutions to a CSP, $\mathrm{sol}\,(C, S)$, is the set of all assignments that are solutions.

A constraint c is *entailed* by the store S iff $\mathrm{cons}\,(S, \mathrm{var}\,(c)) \subseteq c$. Entailed constraints can safely be removed from the CSP since they no longer restrict the set of solutions.

Propagators. To solve a CSP, a constraint programming system uses propagators as implementations of constraints.

A *propagator* is a function p that takes a store S as input and returns a tuple $\langle \mathit{stat}, S'\rangle$, where *stat* is a status message and S' is a new store. The status message will be ignored when only the resulting store is interesting. A propagator p must be *contracting* ($p(S) \leq S$ for all stores S) and *monotonic* (if $S_1 \leq S_2$ then $p(S_1) \leq p(S_2)$ for all stores S_1, S_2). A store S is a *fix-point* of a propagator p iff $p(S) = S$. The status-message indicates if the propagator is entailed (*entailed*) or if it has detected failure ($\bot$). A propagator is *entailed* for a store S iff for all stores $S' \leq S$ it holds that $p(S') = S'$. Entailed propagators can safely be removed from the pool of propagators, since they will do no more pruning. A propagator that reports failure indicates that there are no solutions left, and propagation can be aborted.

A propagator p references variables $\mathrm{var}\,(p) = \{x_1, \ldots, x_n\}$ and is said to *implement* its induced constraint c_p. The induced constraint is defined as the set of assignments that the propagator identifies as solutions:

$$c_p = \left\{ \langle v_1, \ldots, v_n\rangle \mid p(\mathrm{store}\,(\langle v_1, \ldots, v_n\rangle)) = \mathrm{store}\,(\langle v_1, \ldots, v_n\rangle) \right\}$$

For a given constraint c, any propagator p such that $c_p = c$ can be used. Note that many different propagators exist for the very same constraint, typically differing in propagation strength and efficiency.

Constraint programs. Analogously to constraint satisfaction problems, a store and a set of propagators can be combined to form a *constraint program* (CP) $\langle P, S\rangle$. The set of solutions $\mathrm{sol}\,(P, S)$ to a CP is defined as $\mathrm{sol}\,(\{c_p \mid p \in P\}, S)$.

```
Propagate(S)
begin
    Q ← P;
    while Q ≠ ∅ do
        choose and remove p from Q;
        ⟨stat, S'⟩ ← p(S);
        if stat = ⊥ then
            return failure;
        if stat = entailed then
            remove p from P;
        foreach p ∈ ⋃_{x∈dis(S,S')} prop(x) ∩ P do
            Schedule(p);
        S ← S';
    return S;
end

Schedule(p)
begin
    Q ← Q ∪ {p};
end
```

Algorithm 1. Propagator-centered propagation

Variable dependencies. To manage propagation efficiently, a constraint programming system needs to know which propagators may affect which variables, and for which variables a domain change might make a propagator not be at a fix-point. For the purposes of this paper, the set of referenced variables for the propagator can be used as the dependencies.

Dependencies are used in propagation as follows: if a store S is a fix-point of a propagator p, then any store $S' \leq S$ with $\mathrm{var}(p) \cap \mathrm{dis}(S, S') = \emptyset$ is also a fix-point of p. To better characterize how propagators and variables are organized in an implementation, the set of propagators $\mathrm{prop}(x)$ depending on a variable x is defined as $p \in \mathrm{prop}(x)$ if and only if $x \in \mathrm{var}(p)$.

Propagation. Constraint propagation refers to the process of finding the largest mutual fix-point (equivalently, the weakest mutual fix-point with respect to the strength of stores) of the set of propagators from an initial store S that propagation starts from. Since propagators are defined to be monotonic and contracting functions, it is guaranteed that there exists a unique largest mutual fix-point. The cornerstone of a propagation algorithm is to maintain some representation of which propagators might not be at fix-point. Propagator-centered propagation is controlled by the set of propagators still to be propagated (as opposed to variable-centered that maintains a set of modified variables).

Propagator-centered propagation is shown in Algorithm 1. It is assumed that all propagators are contained in the global propagator set P. The global queue Q contains propagators not known to be at fix-point. The choose operation to get the next propagator from Q is left unspecified, but a realistic implementation bases the decision on priority or cost, see for example [21].

Algorithm 1 does not spell out some details. A real system will most probably use events for variable modifications instead of a simple list of dependencies. Furthermore, whether a propagator is at a fix-point or not after it has been propagated should be taken into account to avoid needless re-execution. For a complete discussion of constraint propagation algorithms, see [2], and see [20] for the implementation of these algorithms in constraint programming systems.

3 Groups

A group is an execution manager for a set of propagators. It controls the order of propagation as well as the handling of failure and entailment. The only change to the system is that propagators that belong to a group must be scheduled in their group rather than globally. Running a group is done through a propagator.

Section 3.1 presents a model of groups. Section 3.2 details the implementation of the model. Section 3.3 evaluates the overhead of groups.

3.1 Model

A *group* is an abstraction that supports the operations *schedule* and *propagate*. The first is used for scheduling a propagator that belongs to the group. The second operation is used to run the propagators scheduled in the group.

A relation group (p) is defined for all propagators p. This relation maps a propagator to the group it should be scheduled in. If the propagator is not a member of any group, the relation is empty. To keep the number of concepts small, only propagators can be scheduled and executed by the main propagation loop. Each group g has an associated propagator that is called its *controller propagator*. The controller propagator is given by controller (g). The set of propagators in P that should be scheduled in a group g is given by $\text{prop}(g) = \{p \in P \mid \text{group}(p) = g\}$ (the inverse of the group $(\cdot)$ relation).

A group together with its controller propagator needs to maintain the requirements of a propagator: it should be contracting and monotonic. This must be ensured by all group implementations.

```
Propagate(S)
    ▷ As in Algorithm 1
Schedule(p)
begin
    if group(p) ≠ ∅ then
        group(p).schedule(p);
        Q ← Q ∪ controller(group(p));
    else
        Q ← Q ∪ {p};
end
```

Algorithm 2. Propagator-centered propagation with groups

Algorithm 2 presents the propagation algorithm that supports groups. The only difference from Algorithm 1 is that when scheduling a propagator p, a check is made to see if the propagator belongs to a group. If so, p is scheduled in that group $g = \text{group}(p)$ and controller (g) is added to the global queue.

```
Basic group g
begin
    q : Queue;
    schedule(p) begin
        q.push(p);
    end
    propagate(S) : ⟨Status, Store⟩ begin
        while ¬q.empty do
            p ← q.pop;
            ⟨stat, S'⟩ ← p(S);
            foreach p ∈ ⋃_{x∈dis(S,S')} prop(x) ∩ P do
                Schedule(p);
            if stat = ⊥ then
                return ⟨⊥, S'⟩;
            if stat = entailed then
                remove p from P;
                if prop(g) = ∅ then
                    return ⟨entailed, S'⟩;
            S ← S';
        return ⟨∅, S⟩;
    end
end
```

Algorithm 3. Basic group g with a single flat queue of propagators

Algorithm 3 shows a basic group that maintains a queue of propagators to be executed. This group implements no special behavior except grouping a set of propagators together. Failure of any of the propagators is reported directly. The group is only entailed if all its propagators are entailed (checked by $\text{prop}(g) = \emptyset$). The basic controller propagator runs the group g by executing g.**propagate**$(\cdot)$ and reports entailment and failure accordingly. As will be seen in Sect. 5, this basic group can be used as a building block for more advanced controller propagators.

3.2 Implementation

A group is a simple interface that specifies one function for scheduling a propagator that belongs to that group, `schedule`, and one function for running the propagators scheduled in the group, `propagate`. How scheduling is done is left to the group implementation, as is the method for running the propagators.

To implement the group $(\cdot)$ relation each propagator has a pointer to a group. This means that each propagator needs one extra word of memory. The standard

group is the null group, meaning that the scheduling is done in the normal system queue. If there is a group different from the null group, then the scheduling for that propagator is delegated to the group. This incurs an overhead of one test against null per propagator scheduling. Additionally, one function call per propagator scheduled in a group instead of the global queue needs to be done (the call to g.`schedule`$(\cdot)$).

Execution of a group is managed by the controller propagator. This means that the system does not need to be aware of groups except when scheduling a propagator. A difference from the model is that the way in which the controller propagator for a group is added to the set of propagators to run is programmable; it is done by the `schedule` function of the group. The benefit is that sometimes the controller propagator is guaranteed to be scheduled anyway, and in those cases a back-link to the propagator (represented in the model by controller(g) for a group g) does not need to be maintained. If such a link is desired, the user can add it to the group that needs it with a pointer

The basic group from Algorithm 3 uses the computed set prop$(\cdot)$ to check if there are any propagators left to be scheduled in the group. In a real implementation, computing the set on the fly is not feasible. Instead the cardinality of the set is maintained as a member of a group, and is updated by propagators as they are created and removed.

Optimized implementation of group$(\cdot)$. Implementing the group$(\cdot)$ connection with a pointer in each propagator wastes memory for propagators not belonging to a group. By adding group-scheduled versions of all propagators, only those propagators that belong to a group contains the pointer. Checking if a propagator has a group pointer can be done cheaply using a tag-bit in pointers to it.

The main overhead in this design is the work in adding an extra optional group-scheduled version of each propagator. Additionally, it would preclude moving a propagator dynamically from the general pool into a group.

The implementation of the model has been done in the Gecode system [22] version 3.0.2. The general description of the implementation here is applicable to any propagator-centered system and is by no means specific to Gecode.

3.3 Evaluation

The implementation of groups touches few parts of the core system, and should therefore incur a small overhead. To evaluate this, the queens problem is tried in two variants. Problems `queens-n-*` use the naive model with a quadratic number of binary not-equals constraints. Problems `queens-s-*` use three large alldifferent constraints instead (albeit with naive propagation only to guarantee that both variants have the same search space). The two different versions test the behavior of the system using many small and few but large propagators.

The experiments have been run on an Athlon 64 3500+ with 2GB of RAM running Ubuntu Linux with gcc version 4.2.4. Times are computed as the average of at least 20 runs with a coefficient of deviation of less than 2%. The `queens-*-200` instances were limited to searching 100 000 nodes.

Table 1. Basic overhead of Groups. Systems compared are without groups (`plain`), with groups added but not used (`groups`), and with groups added and used for scheduling (`scheduling`). Time is given in milliseconds and memory in kilobytes allocated.

Problem	`plain` time	`plain` memory	`groups` time	`groups` memory	`scheduling` time	`scheduling` memory
`queens-n-10`	0.16	63	0.16	63	0.17	63
`queens-n-100`	30.38	7 245	28.10	7 885	30.55	7 885
`queens-n-200`	1 876.30	45 779	1 913.25	50 323	2 061.33	50 323
`queens-s-10`	0.05	19	0.05	19	0.05	19
`queens-s-100`	1.21	355	1.22	356	1.14	356
`queens-s-200`	726.10	1 958	715.05	1 958	708.35	1 958

The results are presented in Table 1. The difference in time between `plain` (the base system) and `groups` (groups added but not used) is not significant; it is less than 2% in the worst case. The inevitable overhead associated with scheduling through groups instead of in the normal queue is reasonably low at around 7% in the worst case. The slightly larger memory-overhead for programs with many propagators is due to the fact that each propagator has an additional field for the group it belongs to. If the memory overhead is prohibitive, the memory-optimized design where each propagator pointer is tagged can be used.

4 Controlling Propagation Order

In a modern constraint programming system, the execution order is defined by the system and works on the granularity of single propagators. Propagator groups can be used for debugging by giving a high-level view of the propagation process (Sect. 4.1). For some sets of propagators, the optimal ordering of propagators is known statically. By following this order instead of the generic order chosen by the system, the optimal number of propagation steps can be achieved (Sect. 4.2).

4.1 Debugging

In many constraint models, the high-level constraints that the model is expressed in can each correspond to many smaller concrete propagators in the system used. As demonstrated in [10], grouping these smaller concrete constraints into larger entities that represent the high-level conceptual constraints is beneficial for understanding the propagation-process. If a high-level view of the propagation process is presented, then stepping through propagation is meaningful.

Implementation. Using the basic group presented in Algorithm 3, it is easy to group propagators into hierarchical groups. As long as the propagators in a group are of roughly the same complexity level, the single-queue group works well. If larger groups of different types of propagators is used, an implementation using multiple queues such as presented in [21] could be used.

Table 2. Grouped propagator execution. Model compared are without groups (`plain`) and with groups (`groups`). Time is given in milliseconds and steps are average propagation steps per node.

	`plain`		`groups`	
Problem	time	steps	time	steps
`sudoku-val-5`	5.45	7.55	5.41	4.15
`sudoku-dom-5`	3.27	22.86	3.68	9.75
`sudoku-val-66`	89.38	10.27	99.37	4.68
`sudoku-dom-66`	0.61	314.00	0.96	47.00
`sudoku-val-87`	1 433.37	8.53	1 484.20	4.20
`sudoku-dom-87`	5.00	59.35	6.93	16.57

Evaluation. In the basic constraint programming model for a $n \times n$ Sudoku, there are $3n$ alldifferent constraints. These constraints are composed of three sets that work on rows, columns, and boxes respectively. This division has the interesting property that no two constraints from the same set share a variable.

To illustrate the constraint propagation for a Sudoku problem, it is natural to divide the propagators into the three groups for rows, columns, and boxes. To test this, three instances were run using naive propagation and domain propagation. The numbers refer to the instance-number in the Gecode example. As seen in Table 2, the efficiency of solving a Sudoku is roughly the same, but the number of propagation steps per node is reduced. Thanks to the reduced number of steps, going through the changes between each step becomes feasible.

Using the ability to group the propagators into larger groups, a visualization-system such as the one developed in [14] can be used without getting an overwhelming detail of information during debugging.

4.2 Optimal Propagation Ordering

Decompositions of global constraints that do not sacrifice propagation is an interesting research topic [16, 3, 4, 5]. In some cases (such as the decomposition of the regular constraint into extensional constraints [16]) the decomposition uses a Berge-acyclic propagator graph, which ensures that the local propagation achieves global domain consistency [1]. One benefit of such a decomposition is that the optimal ordering of the propagation is known, with one forward and one backward pass being sufficient to reach a fix-point.

Unfortunately, no constraint programming system supports the specification of propagation order on such a fine-grained level. This means that while the complexity of propagating the decomposition in the optimal order can be calculated, the actual complexity of the decomposition depends on the propagation order that a particular system implements.

Implementation. An *ordered propagation group* g contains a list with the propagators in $\mathrm{prop}(g)$. This list is sorted according to the order in which the propagation should be done. For a Berge-acyclic propagator graph any topological sorting of the propagators works. The controller propagator is the same as for the basic group: it simply runs the group.

On activation, the group does one forward and one backward pass through the list of propagators. When inspected, each propagator is executed until it reaches a fix-point if it has been scheduled. After the two passes, the group is at a fix-point if the propagators form a Berge-acyclic propagator graph.

The implementation does not try to find the set of propagators that should be executed without inspecting all of the propagators since that would complicate the scheduling operation. If the overhead of running through all the propagators is too high for some applications, advisors [12] could be used to record the first and last position that needs to be inspected, delimiting the range of propagators that need to be executed. This is a good compromise, since the scheduling operation can be kept at constant time complexity.

Evaluation. Consider a simple ordering problem on n variables x_i with domain $\{1, \ldots, n\}$. Each pair of consecutive variables is ordered using $x_i < x_{i+1}$. For a system that uses queues for scheduling, as most system do, $O(n^2)$ runs through the propagators are needed to ensure that the variables are assigned their respective values through propagation. The bad behavior is because the scheduling will ensure that only forward-passes are made through the list of propagators. Using a group to schedule the propagators in the optimal order, two passes through the propagators are sufficient to ensure that the variables are assigned.

In Table 3, the simple example is tried for varying number of variables. At small sizes ($n = 10$) there is no measurable overhead to using groups even though the relative benefits in number of steps is smaller. As the sizes grow larger, the complexity difference becomes apparent, with orders of magnitude difference in the time. This example is conservative in that the individual propagators are among the cheapest propagators that exist. The costlier the individual propagators are, the more important it is to use a good propagation ordering.

Table 3. Ordered propagator execution. Model compared are without groups (`plain`) and with groups (`groups`). Time is given in milliseconds and steps are propagation steps. For `group`, the steps are the number of steps inside the controlling group.

	`plain`		`groups`	
Problem	time	steps	time	steps
`order-10`	0.04	36	0.03	18
`order-100`	2.95	4 851	0.39	198
`order-1000`	304.90	498 501	7.31	1 998
`order-5000`	7 743.35	12 492 501	103.35	9 998

5 Controlling Propagator Interaction

Since a group is responsible for executing a propagator, it is possible to control how failure of the propagator is handled. Furthermore, a group also encapsulates entailment, which is dual to failure (it represents failure of the negated constraint). Using these facilities, it is possible to implement general reification (Sect. 5.1) and constructive disjunction (Sect. 5.2) using groups.

5.1 General Reification

A reified constraint $c \leftrightarrow (b = 1)$ reflects if the constraint c holds into a Boolean variable b. Reification is commonly available in constraint programming systems for simple constraints using specialized propagators. For larger and more complex constraints such as alldifferent, the effort of implementation often outweighs the benefit of having a reified version of the constraint.

The basic pattern of propagation for a reified constraint looks as follows.

$$\begin{aligned} c \leftrightarrow (b = 1) \ := \\ c \text{ holds} &\Rightarrow \text{propagate } b = 1 \\ \neg c \text{ holds} &\Rightarrow \text{propagate } b = 0 \\ b = 1 \text{ holds} &\Rightarrow \text{propagate } c \\ b = 0 \text{ holds} &\Rightarrow \text{propagate } \neg c \end{aligned}$$

Implementation. Given a constraint c and a Boolean variable b a simple implementation of the reified constraint $c \leftrightarrow (b = 1)$ can be done by posting the constraint in a basic group g. Instead of using the global store-variables $x = \text{var}(c)$ copies of the variables, x', are made and used for the constraint, giving a modified store S'. The equality relation for the stores S and S' is extended in the obvious way. The controlling propagator proceeds through the following steps:

- Add new domain reductions from x to copied variables x' and run the group.
- If g is failed, set $b = 0$ and report entailment.
- If g is entailed, set $b = 1$. If $S = S'$, report entailment.
- If b is set to one, add local domain reductions to the global store.

Unfortunately, the above implementation of generic reification lacks the possibility to propagate $\neg c$, instead it has to wait for failure or entailment of c. This is not a problem with the approach, but a consequence of the problem of propagating negated constraints and is something that is handled in the same way in general reification in Mozart [18, 19]. To implement reification in Propia [15], both the constraint and the negated constraint must be expressed as CLP clauses.

Evaluation. As a base evaluation of using groups for reification, the standard no-overlaps constraints for perfect square packing are tried. For each pair of squares i and j with coordinates x and y and size s, they do not overlap iff $(x_i + s_i \leq x_j) \vee (x_j + s_j \leq x_i) \vee (y_i + s_i \leq y_j) \vee (y_j + s_j \leq y_i)$. Three versions are tested: using normal reification; using normal reification on copied variables; and

using groups to implement reification. The numbers refer to the instance numbers in the Gecode example. As seen in Table 4 the overhead from using groups for reification is not unreasonable, especially compared with using standard reified propagators on copied variables which is always worse than groups.

Table 4. Reified no-overlap constraint for the perfect square problem. Models compared use normal reification (`reified`), reification on copied variables (`copied`), and an implementation using groups (`group`). Columns v and c indicate the number of variables and the number of reified constraints for no-overlap. Time is given in milliseconds and memory in kilobytes allocated.

			`reified`		`copied`		`group`	
Problem	c	v	time	memory	time	memory	time	memory
`perfsq-0`	840	3360	280.83	4 998	618.35	8 646	493.80	7 366
`perfsq-1`	924	3696	1 530.05	5 254	3 848.30	8 902	3 164.80	7 750
`perfsq-2`	924	3696	388.57	8 326	650.08	12 166	565.12	10 886
`perfsq-3`	1012	4048	461.20	5 510	1 034.44	9 541	865.02	8 134
`perfsq-4`	1012	4048	2 281.11	15 175	3 314.45	19 783	2 960.56	18 436

The Equidistant Frequency Permutation Array (EFPA) problem [9] is a combinatorial design problem. One of the proposed improvements in [9] to the model includes a reified version of alldifferent. Unfortunately, this is not available in most constraint programming systems. Two versions are tested: using a decomposition representing a reified alldifferent; and using a group to implement it.

Table 5. Reified alldifferent for the EFPA problem. Models compared use a decomposition (`decomposed`) and an implementation using groups (`group`) for the reified alldifferent constraint. Time is given in milliseconds.

	`decomposed`		`group`	
$\langle d, \lambda, q, v \rangle$	time	nodes	time	nodes
$\langle 4, 3, 4, 6 \rangle$	0.03	2 192	0.06	1 045
$\langle 4, 4, 4, 8 \rangle$	3.23	20 564	2.32	10 104

As shown in Table 5, the additional pruning from the true reified alldifferent propagator using groups pays of in the number of nodes that need to be explored. The time is slightly improved, although not as much. Using groups, it is possible to try a reified version of alldifferent in a reasonably efficient manner.

5.2 Constructive Disjunction

Given a constraint $c \vee c'$ where c and c' share common variables x, the use of the reified decomposition $c \leftrightarrow (b = 1) \wedge c' \leftrightarrow (b' = 1) \wedge b + b' \geq 1$ sacrifices

propagation. Consider a disjunctive resource constraint between two tasks 1 and 2 with start-times $s_1 \in \{1..10\}$ and $s_2 \in \{1..10\}$ and durations $d_1 = 6$ and $d_2 = 7$. The reified construction $s_1 + d_1 \leq s_2 \leftrightarrow (b_1 = 1) \wedge s_2 + d_2 \leq s_1 \leftrightarrow (b_2 = 1) \wedge b_1 + b_2 \geq 1$ does not propagate any new information. It is not hard to see that the domains could be reduced, giving $s_1 \in \{1..4, 8..10\}$ and $s_2 \in \{1..3, 7..10\}$.

While it is possible to write a specialized propagator to handle disjunctive resources, it may not be cost-efficient to do so. Furthermore, it does not handle the general case of propagating disjunctive constraints. The technique of constructive disjunction [11, 13, 7, 6, 23] can be used to get full propagation. The basic scheme is to add renamed copies of the variables, as in the previous section, and to run the disjunctive constraints on each of the copies. For any given variable x that is shared among the disjuncts, the union of the domains of the variable-copies for the disjuncts is the new domain.

Implementation. For each disjunct c_i, a new copy of the constraint variables $\mathrm{var}(c_i)$ in the store S is made, giving a new store S_i. The propagators for each disjunct c_i are put in a basic group g_i. By putting the disjuncts in separate groups, the status of each disjunct (failure, entailment) can be checked. The controlling propagator proceeds through the following steps:

- For each variable x, add the new domain-reductions to each copied store S_i.
- Run each group g_i that is not yet failed.
- If all groups are failed, report failure.
- If there exists an entailed group g_i where $S_i = S$, report entailment.
- For each variable x shared among all non-failed groups G, set $x = \cup_{g_j \in G} x_j$.

Implementing a constructive disjunction propagator is mostly straightforward. One interesting aspect is that the largest part of the code and logic is related to handling failed and entailed groups so that the (relatively) expensive propagator is not run needlessly and so that no propagation is missed.

Implementing constructive disjunction using groups is similar to the hard-coded implementation from the cc(FD) system [11]. The difference is that with groups the system is kept unaware of constructive disjunction. Programming constructive disjunction in Propia [15] is similar to groups in that the system is not hard-coded for constructive disjunction, while it is different in that it uses nested search to evaluate each disjunct. This is an approach that saves memory and trades it for computation time.

Evaluation. The value of constructive disjunction as a general technique has been investigated previously [23]. To give a basic evaluation of the implementation, the disjunctive resource example from above is tried with normal reification and with constructive disjunction. The branching used is to try the median value. The example is scaled with a factor k representing time granularity on both variables and durations. While this is a small artificial example, it is based on the common serialization constraint. The results in Table 6 show that the use of constructive disjunction gives an important speed-up. For the case where the use of constructive disjunction is desired, groups enable the use without incurring overhead for the system implementer.

Table 6. Simple use of constructive disjunction. Time is given in milliseconds.

	`reified`		`constructive`	
k	time	nodes	time	nodes
100	0.00	402	0.00	18
1 000	0.02	4 002	0.01	182
10 000	0.20	40 002	0.04	1 802
100 000	1.97	400 002	0.18	18 002

6 Conclusions

The addition of groups to a propagator-oriented constraint programming system is a small, simple, and minimal extension that allows several interesting and useful techniques to be implemented at the user-level without any additional support from the system. In particular, groups enable the first implementation of optimal propagation ordering for Berge-acyclic constraint graphs. The implementation is simple, and the overhead of the system is kept low.

The advantage of being able to experiment with defining the order of propagation and to control the execution of propagators opens up for many new interesting topics. For example, combining the analysis of constraint graphs from [8] with groups for optimal propagator ordering would allow optimal propagator ordering for large subsets of the constraints without requiring the user to specify these patterns.

The results show that groups can implement reification for complex constraints without reification-support and constructive disjunction with a moderate overhead.

Acknowledgements. The authors would like to thank Guido Tack and the anonymous reviewers for comments that considerably improved this paper.

References

[1] Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. J. ACM 30(3), 479–513 (1983)
[2] Bessiere, C.: Constraint propagation. In: Rossi, et al. (eds.) [17], ch. 3, pp. 29–84
[3] Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M. (eds.) ECAI, pp. 475–479 (2008)
[4] Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Range and roots: Two common patterns for specifying and propagating counting and occurrence constraints. Artificial Intelligence (2009)
[5] Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G., Walsh, T.: Decompositions of all different, global cardinality and related constraints. In: IJCAI (2009)
[6] Carlson, B., Carlsson, M.: Compiling and executing disjunctions of finite domain constraints. In: ICLP, pp. 117–131 (1995)

[7] Codognet, C., Codognet, P.: Guarded constructive disjunction: Angel or demon? In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976, pp. 345–361. Springer, Heidelberg (1995)
[8] Francis, K., Stuckey, P.J.: Constraint propagation for loose constraint graphs. In: Cho, Y., Wainwright, R.L., Haddad, H., Shin, S.Y., Koo, Y.W. (eds.) SAC, pp. 334–335. ACM, New York (2007)
[9] Gent, I.P., McKay, P., Miguel, I., Nightingale, P., Huczynska, S.: Modelling equidistant frequency permutation arrays in constraints. In: SARA (2009)
[10] Goualard, F., Benhamou, F.: Debugging constraint programs by store inspection. In: Deransart, P., Małuszyński, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 273–297. Springer, Heidelberg (2000)
[11] Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). In: Podelski, A. (ed.) Constraint Programming: Basics and Trends. LNCS, vol. 910, pp. 293–316. Springer, Heidelberg (1995)
[12] Lagerkvist, M.Z., Schulte, C.: Advisors for incremental propagation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 409–422. Springer, Heidelberg (2007)
[13] Müller, T., Würtz, J.: Constructive Disjunction in Oz. In: Krall, A., Geske, U. (eds.) 11. Workshop Logische Programmierung, Technische Universität Wien, September 27-29 (1995)
[14] Paltzer, N.: Debugging constraint propagation. Master's thesis, Saarland University, Germany (March 2008)
[15] Provost, T.L., Wallace, M.: Domain independent propagation. In: FGCS, pp. 1004–1011 (1992)
[16] Quimper, C.-G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
[17] Rossi, F., van Beek, P., Walsh, T.: Handbook of constraint programming (2006)
[18] Schulte, C.: Programming deep concurrent constraint combinators. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 215–229. Springer, Heidelberg (2000)
[19] Schulte, C.: Programming Constraint Services. LNCS (LNAI), vol. 2302. Springer, Heidelberg (2002)
[20] Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In: Rossi, et al. (eds.) [17], ch. 14, pp. 495–526
[21] Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. Transactions on Programming Languages and Systems 31(1), 2:1–2:43 (2008)
[22] The Gecode team. Generic constraint development environment (2006), `http://www.gecode.org`
[23] Würtz, J., Müller, T.: Constructive disjunction revisited. In: Görz, G., Hölldobler, S. (eds.) KI 1996. LNCS, vol. 1137, pp. 377–386. Springer, Heidelberg (1996)