Integrating Strong Local Consistencies into Constraint Solvers

EMN Research Report nb. 09/1/INFO*

Julien Vion, Thierry Petit, and Narendra Jussien

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
julien.vion@emn.fr, thierry.petit@emn.fr, narendra.jussien@emn.fr

Abstract. This report presents a generic scheme for adding strong local consistencies to the set of features of constraint solvers, which is notably applicable to event-based constraint solvers. We encapsulate a subset of constraints into a global constraint. This approach allows a solver to use different levels of consistency for different subsets of constraints in the same model. Moreover, we show how strong consistencies can be applied with different kinds of constraints, including user-defined constraints. As a concrete implementation of our work, we propose a new coarse-grained algorithm for Max-RPC, called Max-RPC^{rm}. Experiments confirm the interest of strong consistencies for Constraint Programming tools.

1 Introduction

This report presents a generic framework for integrating strong local consistencies into Constraint Programming (CP) tools, especially event-based solvers. It is successfully experimented using a new coarse-grained algorithm for Max-RPC.

The most successful techniques for solving problems with CP are based on local consistencies. Local consistencies remove values or assignments that cannot belong to a solution. To enforce a given level of local consistency, *propagators* are associated with constraints. A propagator is complete when it eliminates all the values that cannot satisfy the constraint. One of the reasons for which CP is currently applied with success to real-world problems is that some propagators are encoded through *filtering algorithms*, which exploit the semantics of the constraints. Filtering algorithms are often derived from well-known Operations Research techniques. This provides powerful implementations of propagators.

Many solvers use an AC-5 based propagation scheme [18]. We call them event-based solvers. Each propagator is called according to the events that occur in domains of the variables involved in its constraint. For instance, an event may be a value deleted by another constraint. At each node of the search tree, the pruning is performed within the constraints. The fix point is obtained by

* This work was supported by the ANR french research funding agency, through the CANAR project (ANR-06-BLAN-0383-03).

propagating events among all the constraints. In this context, generalized arc-consistency (GAC) is, *a priori*, the highest level of local consistency that can be enforced (all propagators are complete).

On the other hand, local consistencies that are stronger than GAC [8, 5] require to take into account several constraints at a time in order to be enforced. Therefore, it is considered that such strong consistencies cannot easily be integrated into CP toolkits, especially event-based solvers. Toolkits do not feature those consistencies, and they are not used for solving real-life problems.

This report demonstrates that strong local consistencies are wrongly excluded from CP tools. We present a new generic paradigm to add strong local consistencies to the set of features of constraint solvers. Our idea is to define a *global constraint* [6, 2, 15], which encapsulates a subset of constraints of the model. The strong consistency is enforced on this subset of constraints. Usually a global constraint represents a sub-problem with fixed semantics. It is not the case of our global constraint: It is used to apply a propagation technique on a given subset of constraints, as it was done in [16] in the context of over-constrained problems.

This approach provides some new possibilities compared with the state of the art. A first improvement is the ability to use different levels of consistency for different subsets of constraints in the same constraint model. A second one is to apply strong consistencies to all kinds of constraints, including user-defined constraints or arithmetical expressions. Finally, within the global constraint, it is possible to define any strategy for handling events. One may order events variable per variable instead of considering successively each encaspulated constraint. Event-based solvers generally do not provide such a level of precision.

We experiment our framework with the Max-RPC strong consistency [7], using the Choco CP solver [1]. To preserve as much as possible the practical efficiency, we propose a new coarse-grained algorithm for Max-RPC. This new algorithm exploits backtrack-stable data structures in a similar way to AC-3^{rm} [14], which is one of the most efficient binary arc-consistency algorithms.

2 Background

A *constraint network* $\mathcal{N}$ consists of a set of variables $\mathscr{X}$, a set of domains $\mathscr{D}$, where the domain $\text{dom}(X) \in \mathscr{D}$ of variable X is the finite set of at most d values that variable X can take, and a set $\mathscr{C}$ of e constraints that specify the allowed combinations of values for given subsets of variables. An instantiation I is a set of couples variable/value, (X, v), denoted X_v. I is *valid* iff for any variable X involved in I, $v \in \text{dom}(X)$. A *relation* R of arity k is any set of instantiations of the form $\{X_a, Y_b, \ldots, Z_c\}$, where $a, b, \ldots, c$ are values from a given universe. A *constraint* C of arity k is a pair $(\text{scp}(C), \text{rel}(C))$, where $\text{scp}(C)$ is a set of k variables and $\text{rel}(C)$ is a relation of arity k. $I[X]$ denotes the value of X in the instantiation I. For binary constraints, C_{XY} denotes the constraint *s.t.* $\text{scp}(C) = \{X, Y\}$. Given a constraint C, an instantiation I of $\text{scp}(C) = \{X, \ldots, Z\}$ (or of a superset of $\text{scp}(C)$, considering only the variables in $\text{scp}(C)$), *satisfies* C iff $I \in \text{rel}(C)$. We say that I is *allowed* by C. A *solution* of a constraint

network $\mathcal{N}(\mathscr{X},\mathscr{D},\mathscr{C})$ is an instantiation I_S of all variables in $\mathscr{X}$ s.t. (1.) $\forall X \in \mathscr{X}, I_S[X] \in \mathrm{dom}(X)$ (I_S is *valid*), and (2.) I_S satisfies (is *allowed* by) all the constraints in $\mathscr{C}$.

2.1 Local consistencies

Definition 1 (Support, Arc-Consistency). *Let C be a constraint and $X \in \mathrm{scp}(C)$. A* **support** *for a value $a \in \mathrm{dom}(X)$ w.r.t. C is a valid instantiation $I \in \mathrm{rel}(C)$ s.t. $I[X] = a$. Value $a \in \mathrm{dom}(X)$ is* **arc-consistent** *w.r.t. C iff it has a support in C. $\mathrm{dom}(X)$ is arc-consistent w.r.t. C iff $\forall a \in \mathrm{dom}(X)$, a has a support w.r.t. C. C is arc-consistent iff $\forall X \in \mathrm{scp}(C)$, $\mathrm{dom}(X)$ is arc-consistent.*

In the case of a binary constraint, *i.e.*, between X and Y, the support $I = \{X_a, Y_b\}$ of a value X_a can simply be characterized by the value Y_b. We say that Y_b supports X_a. In the remaining of this report we will call arc-consistency AC when constraints are binary, and GAC when constraints can have any arity.

Definition 2 (Closure). *Let $\mathcal{N}(\mathscr{X},\mathscr{D},\mathscr{C})$ be a constraint network, Φ a local consistency (e.g.,* AC*) and $\mathcal{C}$ a set of constraints $\subseteq \mathscr{C}$. $\Phi(\mathscr{D},\mathcal{C})$ is the* closure *of $\mathscr{D}$ for Φ on $\mathcal{C}$, i.e. the set of domains obtained from $\mathscr{D}$ where $\forall X$, all values $a \in \mathrm{dom}(X)$ that are not Φ-consistent w.r.t. a constraint in $\mathcal{C}$ have been removed.*

For GAC and for most consistencies, the closure is unique. In CP systems, a *propagator* is associated with each constraint to enforce GAC or weaker forms of local consistencies. On the other hand, local consistencies stronger than GAC [8, 5] require to take into account more than one constraint at a time to be enforced. This fact have made them excluded from most of CP solvers, until now.

2.2 Strong local consistencies

This report focuses on domain filtering consistencies [8], which only prune values from domains and leave the structure of the constraint network unchanged.

Firstly, w.r.t. binary constraint networks, as it is mentioned in [5], (i, j)-consistency [10] is a generic concept that captures many local consistencies. A binary constraint network is (i, j)-consistent iff it has non-empty domains and any consistent instantiation of i variables can be extended to a consistent instantiation involving j additional variables. Thus, AC is $(1, 1)$-consistency.

A binary constraint network $\mathcal{N}$ that has non empty domains is *Path Consistent* (PC) iff it is $(2, 1)$-consistent. It is *Path Inverse Consistent* (PIC) [11] iff it is $(1, 2)$-consistent. It is *Restricted Path Consistent* (RPC) [3] iff it is $(1, 1)$-consistent and for all values a that have a single consistent extension b to some variable, the pair of values (a, b) forms a $(2, 1)$-consistent instantiation. $\mathcal{N}$ is *Max-Restricted Path Consistent* (Max-RPC) [7] iff it is $(1, 1)$-consistent and for each value X_a, and each variable $Y \in \mathscr{X} \backslash X$, one consistent extension Y_b of X_a is $(2, 1)$-consistent (that is, can be extended to any third variable). It is *Singleton Arc-Consistent* (SAC) [8] iff each value is SAC, and a value X_a is SAC if the

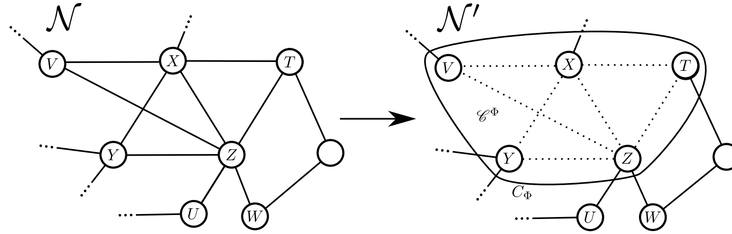

Fig. 1. A strong consistency global constraint C_{Φ}, used to enforce the strong local consistency on a subset of constraints $\mathscr{C}^{\Phi}$. $\mathcal{N}'$ is the new network obtained when replacing $\mathscr{C}^{\Phi}$ by the global constraint.

subproblem built by assigning a to X can be made AC (the principle is very close to shaving, except that here the whole domains are considered). Finally, $\mathcal{N}$ is *Neighborhood Inverse Consistent* (NIC) [11] iff any consistent assignment of value a to variable $X \in \mathscr{X}$ can be extended to a consistent assignment of all the variables in the neighborhood of X within the constraint graph.

Concerning non binary constraint networks, relational arc-consistency and path consistency [9] (relAC and relPC) provide concepts useful to extend local consistencies defined for binary constraint networks to the non-binary case. A constraint network $\mathcal{N}$ is relAC iff any consistent assignment for all but one of the variables in a constraint can be extended to the last variable, so as to satisfy the constraint. $\mathcal{N}$ is relPC iff any consistent assignment for all but one of the variables in a pair of constraints can be extended to the last variable so as to satisfy both constraints. From these notions, new domain filtering consistencies for non-binary constraints inspired by the definitions of RPC, PIC and Max-RPC were proposed in [5]. Moreover, some interesting results were obtained using pairwise consistency. A constraint network $\mathcal{N}$ that has non empty domains is *Pairwise Consistent* (PWC) [12] iff it has non-empty relations and any consistent tuple of a constraint can be consistently extended to any other constraint that intersects with this. One may apply both PWC and GAC. $\mathcal{N}$ is *Relationally Path Inverse Consistent* (relPIC) iff it is relationally $(1, 2)$-consistent. It is *Pairwise Inverse Consistent* (PWIC) [17] iff for each value X_a, there is a support for a w.r.t. all constraints involving X, s.t. the supports in all constraints that overlap on more variables than X have the same values.

3 A Global constraint for Domain filtering consistencies

This section presents an object-oriented generic scheme for integrating domain filtering consistencies in constraint solvers, and its specialization for Max-RPC. Given a local consistency Φ, the principle is to deal with the subset $\mathscr{C}^{\Phi}$ of constraints on which Φ should be applied, within a new global constraint C_{Φ} added to the constraint network. Constraints in $\mathscr{C}^{\Phi}$ are connected to C_{Φ} instead of being included into the initial constraint network $\mathcal{N}$ (see Figure 1). In this way,

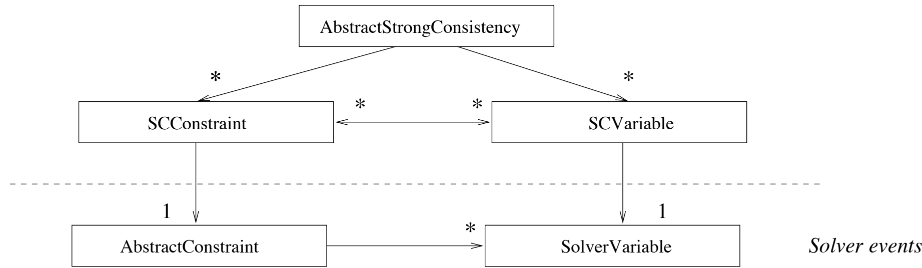

Fig. 2. Diagram of the integration of strong local consistencies into event-based solvers. Arrows "→" depict aggregation relations.

events related to constraints in $\mathscr{C}^{\Phi}$ are handled in a closed world, independently from the propagation queue of the solver.

3.1 A generic scheme

As it is depicted by Figure 2, `AbstractStrongConsistency` is the abstract class that will be concretely specialized for implementing C_{Φ}, the global constraint that enforces Φ. The constraint network corresponding to $\mathscr{C}^{\Phi}$ is stored within this global constraint. In this way, we obtain a very versatile framework to implement any consistency algorithm within the event-based solver.

We encapsulate the constraints and variables of the original network in order to rebuild the constraint graph involving only the constraints in $\mathscr{C}^{\Phi}$, thanks to `SCConstraint` (Strong Consistency Constraint) and `SCVariable` (Strong Consistency Variable) classes. In Figure 1, in $\mathcal{N}'$ all constraints of $\mathscr{C}^{\Phi}$ are disconnected from the original variables of the solver. Variables of the global constraint are encapsulated in `SCVariable`s, and the constraints in `SCConstraint`s. In $\mathcal{N}'$, variable Z is connected to the constraints C_{UZ}, C_{WZ} and C_{Φ} from the point on view of the solver. Within the constraint C_{Φ}, the `SCVariable` Z is connected to the dotted `SCConstraint`s towards the `SCVariable`s T, V, X and Y.

Mapping the constraints. We need to identify a lowest common denominator among local consistencies, which will be implemented using the services provided by the constraints of the solver. In Figure 2, this is materialized by the abstract class `AbstractSolverConstraint`. Within solvers, and notably event-based solvers, constraints are implemented with *propagators*. While some consistencies such as SAC can be implemented using those propagators, this is not true for most other consistencies. Indeed, the generic concepts that capture those consistencies are (i, j)-consistency, relAC and relPC (see section 2.2). Therefore, they rather rely on the notion of allowed and valid instantiations, and it is required to be able to iterate over these. Moreover, algorithms that seek optimal worst-case time complexities memorize which instantiations have already been considered. This usually requires that a given iterator over the instantiations of a constraint always delivers the instantiations in the same order (generally lexicographic), and the ability to start the iteration from any given instantiation.

Algorithm 1: nextSupport(C, I_{fixed}, I): Instantiation

```
1 I_next ← nextValid(scp(C)\ scp(I_fixed), I) ;
2 while I_next ≠ ⊥ ∧ ¬check(C, I_next ∪ I_fixed) do
3     I_next ← nextValid(scp(C)\ scp(I_fixed), I_next) ;
4 return I_next;
```

Algorithm 2: firstSupport(C, I_{fixed}): Instantiation

```
1 I ← firstValid(scp(C)\ scp(I_fixed)) ;
2 if check(C, I ∪ I_fixed) return I;
3 else return nextSupport(C, I_fixed, I);
```

Iterators can be implemented using *constraint checkers*. A constraint checker checks whether a given instantiation is allowed by the constraint or not. On the other hand, for many constraints, more efficient iterators can be devised. Thus, iterating over the allowed and valid instantiations is done within the methods `firstSupport` and `nextSupport`. `AbstractSolverConstraint`, a subclass of the abstract constraint class of the solver, specify the two methods.

Generic iterators. The `firstSupport` and `nextSupport` services are not usually available in most constraint solvers. We propose a generic implementation of these functions that exclusively rely on the *constraint checkers*. This implementation is wrapped in an `Adapter` class that specializes the required `AbstractSolverConstraint` superclass, and handles any solver constraint with a constraint checker, as depicted by Figure 3. In this way, no modification is made on the constraints of the solver. Our implementation of `firstSupport` and `nextSupport` is given on Algorithms 1 and 2. The time complexity is in $O(d^{|\mathrm{scp}(C)|-|I_{fixed}|})$. It is incremental: the whole process of iterating over all allowed and valid instantiations by calling `firstSupport` and then consistently `nextSupport` with the same I_{fixed} parameter has the same complexity. These generic functions will be mostly suitable for loose constraints of low arity.

Specialized iterators. For some constraints, *e.g.,* user-defined constraints, powerful ad-hoc algorithms for `firstSupport` and `nextSupport` functions can be provided (*e.g.*, arithmetical constraints, or positive table constraints [4]). These algorithms may not use constraint checkers. `AbstractSolverConstraint` is a specialization of `SolverConstraint` (see Figure 3), so, for this purpose, it is sufficient to specialize `AbstractSolverConstraint`.

Using Propagators. Some strong consistencies such as Path Consistency may be implemented by directly using the propagators of the constraints [13]. Our framework also allows these implementations, since the original propagators of the constraints are still available.

Mapping the variables. Mapping the variables is simpler, as our framework only requires basic operations on domains, *i.e.*, iterate over values in the current

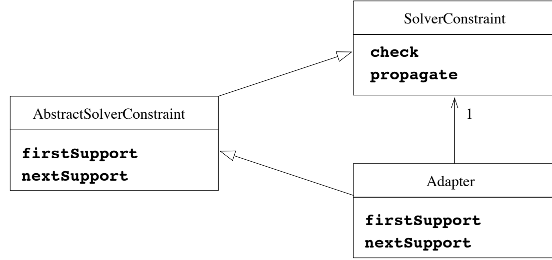

Fig. 3. A generic implementation of support iterator functions, given the constraints provided by a solver. Arrows "—▷" depict specialization relations.

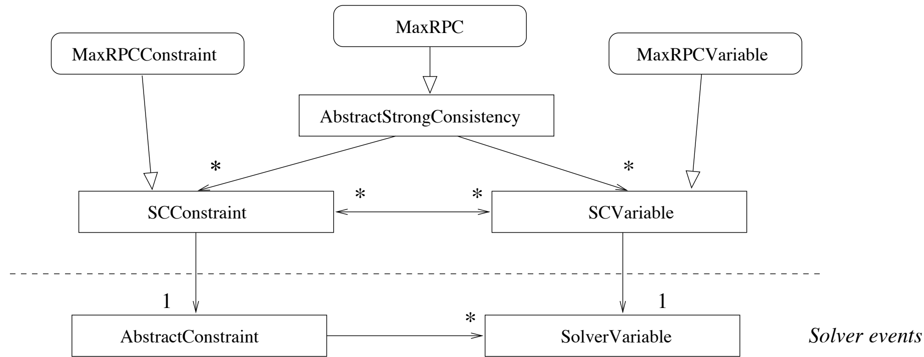

Fig. 4. Diagram of the integration of MaxRPC into event-based solvers.

domain and remove values. Class `SCVariable` is used for representing the constraint subnetwork $(\text{scp}(C_\Phi), \mathscr{D}, \mathscr{C}^\Phi)$. A link is kept with the solver variable for operation on domains.

Variable degree-based heuristics. Some very popular variable ordering heuristics for binary constraints networks, such as Brelaz, *dom/ddeg* or *dom/wdeg*, rely on the structure of the constraint graph in order to select the next variable to instantiate. Since constraints in $\mathscr{C}^\Phi$ are not connected to the model, they are no longer taken into account by the heuristics of the solver. To overcome this issue, we made the heuristics ask directly for the score of a variable to the `AbstractStrongConsistency` constraints that imply this variable. The global constraint is thus able to compute the corresponding dynamic (weighted) degrees of each variable within their subnetwork $\mathscr{C}^\Phi$.

3.2 A concrete specialization: Max-RPC

Figure 4 depicts the specialization of our framework to a particular domain filtering consistency for binary networks, Max-RPC [7]. The class `MaxRPC` defines the

global constraint that will be used in constraint models. It extends the abstract class `AbstractStrongConsistency` to implement the propagation algorithm of Max-RPC. Moreover, implementing Max-RPC requires to deal with 3-cliques in the constraint graph, to check extensions of a consistent instantiation to any third variable. `SCConstraint` and `SCVariable` classes are extended to efficiently manipulate 3-cliques.

4 A new coarse grained algorithm for Max-RPC

This section presents Max-RPC^{rm}, a new *coarse-grained* algorithm for Max-RPC, used in section 5 to experiment our approach. This algorithm exploits backtrack-stable data structures inspired from AC-3^{rm} [14]. rm stands for *multidirectional residues*; a residue is a support which has been stored during the execution of the procedure that proves that a given value is AC. During forthcoming calls, this procedure simply checks whether that support is still valid before searching for another support from scratch. The data structures are stable on backtrack (they do not need to be reinitialized nor restored), hence a minimal overhead on the management of data. Despite being theoretically suboptimal in the worst case, Lecoutre & Hemery showed in [14] that AC-3^{rm} behaves better than the optimal algorithm in most cases.

4.1 The algorithm

Coarse-grained means that the propagation in the algorithm is managed on a variable or a constraint level, whereas fine-grained algorithms such as AC-6 or GAC-schema manage the propagation on a value level. Propagation queues for coarse-grained algorithms are lighter, can be implemented very efficiently and do not require to manage extra data structures for recording which values a given instantiation supports.

Algorithms 3 to 6 describe Max-RPC^{rm}. Lines 6-8 of Algorithm 3 and Lines of 5-8 of Algorithm 5 are added to a standard AC-3^{rm} algorithm.

Algorithm 3 contains the main loop of the algorithm. It is based on a queue containing variables that have been modified (i.e. have lost some values), which may cause some values in the neighbor variables to lose their supports. In the example depicted on Figure 1 (considering only constraints in $\mathscr{C}^{\Phi}$), if the variable X is modified, then the algorithm must check whether all values in T still have a support w.r.t. the constraint C_{XT}, all values in V have a support w.r.t. C_{XV}, and so on for Y and Z. This is performed by Lines 4-5 of Algorithm 3. The `revise` function depicted in Algorithm 5 controls the existence of such supports. It removes the value and returns **true** iff it does not have any (or **false** if the value has not been removed).

The modified variable that has been picked is also likely to have caused the loss of PC supports for constraints situated on the opposite side of the 3-clique. In Figure 1, if X is modified, then the supports of V and Z w.r.t. C_{VZ}, the

Algorithm 3: MaxRPC($P = (\mathscr{X}, \mathscr{C}), \mathscr{Y}$)

$\mathscr{Y}$: the set of variables modified since the last call to MaxRPC

```
1 𝒬 ← 𝒴 ;
2 while 𝒬 ≠ ∅ do
3   pick X from 𝒬 ;
4   foreach Y ∈ 𝒳 | ∃C_XY ∈ 𝒞 do
5     foreach v ∈ dom(Y) do if revise(C_XY, Y_v, true) then 𝒬 ← 𝒬 ∪ {Y};
6   foreach (Y, Z) ∈ 𝒳² | ∃(C_XY, C_YZ, C_XZ) ∈ 𝒞³ do
7     foreach v ∈ dom(Y) do if revisePC(C_YZ, Y_v, X) then 𝒬 ← 𝒬 ∪ {Y};
8     foreach v ∈ dom(Z) do if revisePC(C_YZ, Z_v, X) then 𝒬 ← 𝒬 ∪ {Z};
```

Algorithm 4: revisePC(C_{YZ}, Y_a, X): boolean

Y: the variable to revise because PC supports in X may have been lost

```
1 if pcRes[C_YZ, Y_a][X] ∈ dom(X) then return false ;
2 b ← findPCSupport(Y_a, Z_res[C_YZ, Y_a], X) ;
3 if b = ⊥ then return revise(C_YZ, Y_a, false) ;
4 pcRes[C_YZ, Y_a][X] ← b; return false;
```

supports of Y and Z w.r.t. C_{YZ} and the supports of T and Z w.r.t. C_{TZ} need to be checked. This is the purpose of Lines 6-8 of Algorithm 3 and of the function `revisePC` (Algorithm 4).

Algorithm 5 iterates over the supports (X_b) of the value to revise (Y_a), on Lines 2 and 13, in search of a PC instantiation $\{X_b, Y_a\}$. The Path Consistency of the instantation is checked by calling `findPCSupport` (Algorithm 6) on each variable Z that forms a 3-clique with X and Y. `findPCSupport` returns either a support of the instantiation $\{X_b, Y_a\}$ in Z, or the special value $\perp$ if none can be found. Iff no PC support for Y_a can be found, the value is removed and the function returns **true**.

Residues. The `revise` function firstly checks the validity of the residue (Line 1). Residues are stored in the global data structure $res[C, X_a]$, which has an $O(ed)$ space complexity.

The algorithm also makes use of residues for the PC supports, stored in the structure $pcRes$ with an $O(cd)$ space complexity (c is the number of 3-cliques in the constraint network). The idea is to associate the residue found by the `revise` function with the found PC value for each third variable of the 3-clique. In this way, at the end of the processing, $(X_a, res[C_{XY}, X_a], pcRes[C_{XY}, X_a][Z])$ forms a 3-clique in the micro-structure of the constraint graph for all 3-cliques (X, Y, Z) of the constraint network and for all $a \in \text{dom}(X)$.

In the example depicted on Figure 5, at the end of the processing we have $res[C_{XY}, X_a] = b$, $pcRes[C_{XY}, X_a][Z] = a$, $pcRes[C_{XY}, Y_a][Z'] = a$, and so on. The algorithm exploits the bi-directionnality of the constraints: if Y_b is a support

Algorithm 5: revise(C_{XY}, Y_a, *supportIsPC*): boolean

```
   Y_a: the value of Y to revise against C_XY – supports in X may have been lost
   supportIsPC: false if one of pcRes[C_XY, Y_a] is no longer valid
1  if supportIsPC ∧ res[C_XY, Y_a] ∈ dom(X) then return false ;
2  b ← firstSupport(C_XY, {Y_a})[X] ;
3  while b ≠ ⊥ do
4  |  PConsistent ← true ;
5  |  foreach Z ∈ 𝒳 | (X, Y, Z) form a 3-clique do
6  |  |  c ← findPCSupport(Y_a, X_b, Z) ;
7  |  |  if c = ⊥ then PConsistent ← false ; break;
8  |  |_ currentPcRes[Z] ← c ;
9  |  if PConsistent then
10 |  |  res[C_XY, Y_a] ← b ; res[C_XY, X_b] ← a ;
11 |  |  pcRes[C_XY, Y_a] ← pcRes[C_XY, X_b] ← currentPcRes ;
12 |  |_ return false ;
13 |_ b ← nextSupport(C_XY, {Y_a}, {X_b, Y_a})[X] ;
14 remove a from dom(Y) ; return true ;
```

Algorithm 6: findPCSupport(X_a, Y_b, Z): value

```
1  c_1 ← firstSupport(C_XZ, {X_a})[Z] ; c_2 ← firstSupport(C_YZ, {Y_b})[Z] ;
2  while c_1 ≠ ⊥ ∧ c_2 ≠ ⊥ ∧ c_1 ≠ c_2 do
3  |  if c_1 < c_2 then c_1 ← nextSupport(C_XZ, {X_a}, {X_a, Z_{c_2−1}})[Z] ;
4  |_ else c_2 ← nextSupport(C_YZ, {Y_b}, {Y_b, Z_{c_1−1}})[Z] ;
5  if c_1 = c_2 then return c_1 ; else return ⊥ ;
```

for X_a with $\{Z_a, Z'_a\}$ as PC supports, then X_a is also a support for Y_b with the same PC supports. This is done on Lines 10-11 of Algorithm 5.

If a lost PC support is detected on Line 1 of `revisePC`, then an alternative support is searched. If none can be found, then the current support of the current value is no longer PC, and another one must be found. This is done by a call to `revise` on Line 3 of Algorithm 4.

4.2 Light-MaxRPC^{rm}

Considering the memory overhead caused by the *pcRes* data structure and the calls to the `revisePC` function, we propose to modify Algorithm 3 by removing the **foreach do** loop on Lines 6-8. The `revisePC` function and *pcRes* data structure are no longer useful and can be removed, together with Lines 8 and 11 of Algorithm 5. The obtained algorithm achieves an approximation of Max-RPC, which is stronger than AC. It ensures that all the values that were not Max-RPC before the call to Light-MaxRPC^{rm} will be filtered.

The obtained consistency is not monotonous and will depend on the order in which the modified variables are picked from $\mathcal{Q}$. However, experiments we

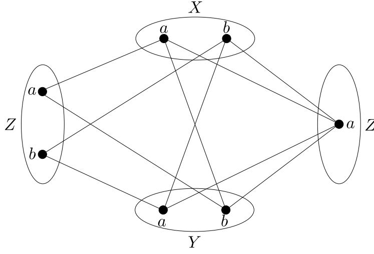

Fig. 5. Example with two 3-cliques (microstructure)

conducted (see Section 5) show empirically that the filtering power of Light-MaxRPC is only slightly weaker than that of Max-RPC on random problems, despite the significant gains in space and time complexities. We even noticed that the different choices made by the heuristics as the result of the different pruning may have more impact on the size of the search tree than what would be earned by the few additional value removals.

4.3 Complexity issues

We use these additional notations: c is the number of 3-cliques in the constraint graph ($c \leq \binom{n}{3} \in O(n^3)$), g is the maximum degree of a variable and s is the maximum number of 3-cliques that share the same single constraint in the constraint graph. If the constraint graph is not empty (at least two variables and one constraint), we have the following relation: $s < g < n$. The complexities are devised in terms of constraint checks (assumed in constant time).

Proposition 1. *After an initialization phase in* $O(eg)$*,* MaxRPC^{rm} *has a worst-time complexity in* $O(ed^3 + csd^4)$*.*

Proof (Sketch). The initialization phase consists in detecting and linking all 3-cliques to their associated constraints and variables, which can be done in $O(eg)$.

The main loop of the algorithm depends on the variable queue $\mathcal{Q}$. Since variables are added to $\mathcal{Q}$ when they are modified, they can be added at most d times in the queue, which implies that the main loop can be performed $O(nd)$ times. This property remains true when the algorithm is called multiple times, removing one value from the domain of one variable every time. The algorithm is incremental, especially when maintaining Max-RPC in a systematic search algorithm. We consider separately the two parts of the main loop.

1. the **foreach do** loop at Lines 4-6 of Algorithm 3.
   This loop can be performed $O(g)$ times. Since in the worst case, the whole network is explored thoroughly in an homogeneous way, it is amortized with the $O(n)$ factor of the main loop in a global $O(e)$ complexity. The **foreach do** loop at Lines 5-6 involves $O(d)$ calls to `revise` (total $O(ed^2)$ revises).
   `revise` (Algorithm 5) first calls `firstSupport`, which has a complexity of $O(d)$ using Algorithm 2 (without any assumption on the nature of the

constraint). The **while do** loop can be performed $O(d)$ times. Calls to `nextSupport` (Line 15) are part of the loop. The **foreach do** loop on Lines 5-10 can be performed $O(s)$ times, and involves a call to `findPCSupport` in $O(d)$. Thus, `revise` is in $O(d+sd^2)$. The global complexity of this first part is thus $O(ed^3 + esd^4)$. The $O(es)$ factor is amortized to $O(c)$, thus a final result in $O(ed^3 + cd^4)$.
2. the **foreach do** loop at Lines 7-11 of Algorithm 3.
The number of turns this loop can perform is amortized with the main loop to an $O(cd)$ factor. Each turn executes $O(d)$ calls to `revisePC`, whose worst-case time complexity is capped by a call to `revise` on Line 3 of Algorithm 4. This part of the algorithm is thus in $O(cd^2.(d + sd^2)) = O(csd^4)$.

The init phase with the two parts result in a complexity of $O(eg+ed^3+csd^4)$.

On a complete constraint graph, time complexity is $O(n^4d^4)$. It can be compared to the lower bound of $O(eg + ed^2 + cd^3)$ ($O(n^3d^3)$ for a complete graph) for an optimal Max-RPC algorithm. If `revise` is called due to the removal of a value that does not appear in any support, its complexity falls down to $O(sd)$. In practice, this happens very regularly, which explains the good practical behavior of the algorithm. As Light-MaxRPC^{rm} skips the 2nd part of the algorithm, its time complexity is $O(eg+ed^3+cd^4)$. Note that with all variants of the algorithm, the init phase in $O(eg)$ is performed previously to the first call to Algorithm 3 and only once when maintaining the Max-RPC property throughout the search.

Finally, we can compare these complexities for enforcing a level of consistency strictly stronger than AC with AC-3^{rm} (in $O(n^2d^3)$ on a complete graph).

5 Experiments

We implemented the diagram of Figure 4 in Choco [1], using the new algorithm for Max-RPC described in section 4. In our experiments, MaxRPC^{rm} and its light variant are compared to AC-3^{rm}. On the figures, each point is the median result over 50 generated binary random problem of various characteristics. A binary random problem is characterized by a quadruple (n, d, γ, t) whose elements respectively represent the number of variables, the number of values, the density[1] of the constraint graph and the tightness[2] of the constraints.

Pre-processing. Figure 6 compares the time and memory used for the initial propagation on rather large problems (200 variables, 30 values). In our experiments, only constraints that form a 3-clique are mapped to the global constraint. A low density leads to a low number of 3-cliques, hence experimental results are coherent with theoretical complexities. Compared with Max-RPC, the low space requirements of Light-Max-RPC^{rm} as well as its nice behaviour clearly appear. Compared with results in [8], embedding a strong consistency into a global constraints present no major overhead. Choco's AC-3^{rm} uses lazy data structures allocation, which explains the drop in memory usage after the threshold point.

[1] The density is the proportion of constraints in the graph w.r.t. the maximal number of possible constraints, i.e. $\gamma = e/\binom{n}{2}$.
[2] The tightness is the proportion of instantiations forbidden by each constraint.

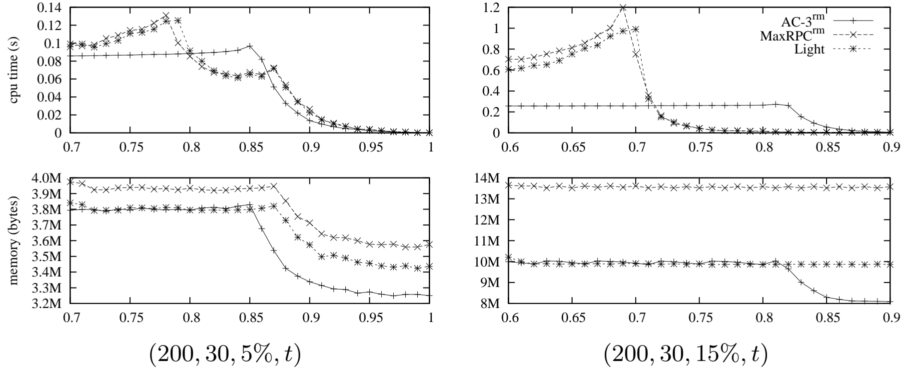

Fig. 6. Initial propagation: cpu time and memory against tightness on homogeneous random problems (200 variables, 30 values).

		AC	MaxRPC	Light	AC+MaxRPC	AC+Light
$(35, 17, 44\%, 31\%)$	*cpu (s)*	**6.1**	25.6	11.6	non	non
	nodes	21.4k	5.8k	8.6k	applicable	applicable
$(105, 20, 5\%, 65\%)$	*cpu (s)*	20.0	19.4	**16.9**	non	non
	nodes	38.4 k	20.4 k	19.8 k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	96.8	167.2	103.2	90.1	**85.1**
$+(105, 20, 5\%, 65\%)$	*nodes*	200.9k	98.7k	107.2k	167.8k	173.4k
$(110, 20, 5\%, 64\%)$	*cpu (s)*	73.0	60.7	**54.7**	non	non
	nodes	126.3k	54.6k	56.6k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	408.0	349.0	272.6	284.1	**259.1**
$+(110, 20, 5\%, 64\%)$	*nodes*	773.0k	252.6k	272.6k	308.7k	316.5k

Table 1. Mixing two levels of consistency in the same model

Max-RPC vs AC. Figure 7 depicts experiments with a systematic search algorithm, where the various levels of consistency are maintained throughout search. The variable ordering heuristic is *dom/ddeg* (the process of weighting constraints with *dom/wdeg* is not defined when more than one constraint lead to a domain wipeout). We use the problem $(105, 20, 5\%, t)$ as a reference (top left graphs) and increase successively the number of values (top right), of variables (bottom left) and density (bottom right). Results in [7] showed that maintaining Max-RPC in a dedicated solver was interesting for large and sparse problems, compared with maintaining AC. However, in [7], AC algorithms that were used are now obsolete compared with AC-3^{rm}. Our results show that encoding our new algorithms for Max-RPC, within a global constraint, is competitive both w.r.t. nodes and time compared with AC-3^{rm}, on large and sparse problems.

Mixing local consistencies. Table 1 shows the effectiveness of the new possibility of mixing two levels of consistency within the same model. The first row corresponds to the median results over 50 instances of problems $(35, 17, 44\%, 31\%)$ forced to be satisfiable. Conversely, the seeds of $(105, 20, 5\%, 65\%)$ are selected so that all problems are unsatisfiable. The first problem is better resolved by using

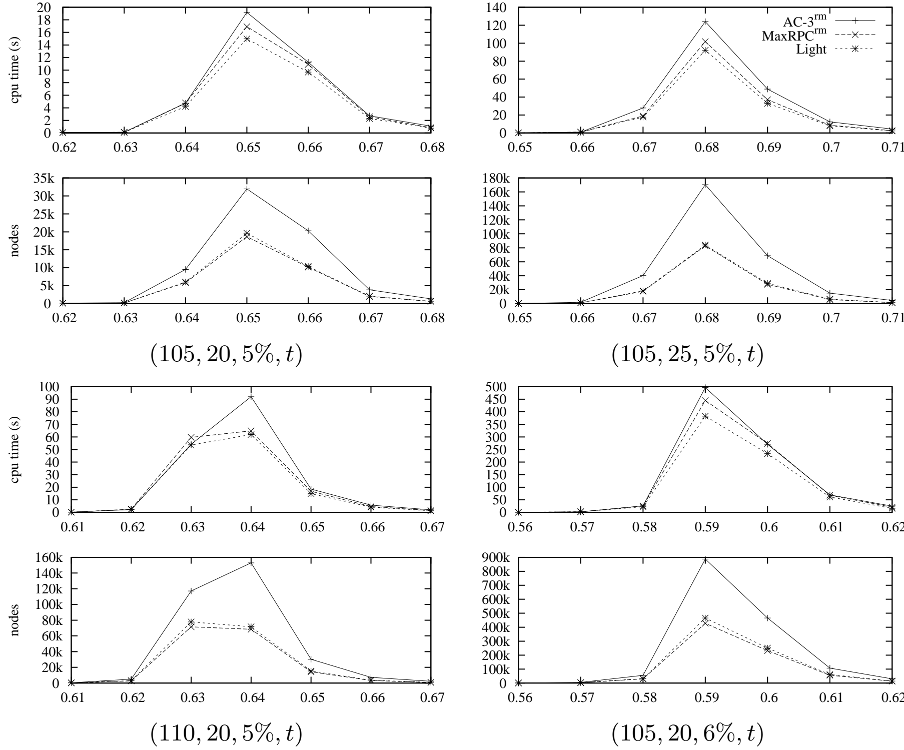

Fig. 7. Full search: cpu time and nodes against tightness on homogeneous random problems (105-110 variables, 20-25 values).

AC-3^{rm} while the second one shows better results with Max-RPC. Third row corresponds to instances where two problems are concatened and linked with a single additional loose constraint. On the last two columns, we maintain AC on the denser part of the model, and Max-RPC^{rm} on the rest. The $dom/ddeg$ variable ordering heuristic will lead the search algorithm to solve firstly the denser, satisfiable part of problem, and then thrashes as it proves that the second part of the model is unsatisfiable. Mixing the two consistencies entails a faster solving, which emphasizes the interest of our approach. The last two rows present the results with larger problems.

6 Conclusion & Perspectives

This report presented a generic scheme for adding strong local consistencies to the set of features of constraint solvers. This technique allows a solver to use different levels of consistency for different subsets of constraints in the same model. The interest of this feature is validated by our experiments. Moreover, strong consistencies can be applied with different kinds of constraints, including

user-defined constraints. Finally, we proposed Max-RPC^{rm}, a new coarse-grained algorithm for Max-RPC.

Future works include the practical use of our framework with other strong local consistencies, as well as a study of some criteria for decomposing a constraint network, in order to automatize the use of different levels of consistency for different subsets of constraints. Further, since a given local consistency can be applied only on a subset of constraints, a perspective opened by our work is to identify specific families of constraints for which a given strong consistency can be achieved more efficiently.

References

1. Choco: An open source Java CP library. `http://choco.emn.fr/`, 2008.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, SICS, 2005.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proc. IEEE-CAIA'95*, 1995.
4. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proc. IJCAI'97*, pages 398–404, 1997.
5. C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.
6. C. Bessière and P. van Hentenryck. To be or not to be... a global constraint. In *Proc. CP'03*, pages 789–794. Springer, 2003.
7. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. CP'97*, pages 312–326, 1997.
8. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
9. R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.
10. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
11. E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proc. AAAI'96*, pages 202–208, 1996.
12. P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation. In *Proc. of IEEE International Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.
13. C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proc. CP'07*, pages 438–452, 2007.
14. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proc. IJCAI'07*, pages 125–130, 2007.
15. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. AAAI'94*, pages 362–367, 1994.
16. J-C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An original constraint based approach for solving over constrained prolems. In *Proc. CP'00*, pages 543–548, 2000.
17. K. Stergiou and T. Walsh. Inverse consistencies for non-binary constraints. In *Proc. ECAI'06*, volume 6, pages 153–157, 2006.
18. P. van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.