Integrating Strong Local Consistencies into Constraint Solvers[*]

Julien Vion[12], Thierry Petit, and Narendra Jussien[3]

[1] Univ Lille Nord de France, F-59500 Lille, France
[2] UVHC, LAMIH FRE CNRS 3304, F-59313 Valenciennes, France
`julien.vion@univ-valenciennes.fr`

[3] École des Mines de Nantes,
LINA UMR CNRS 6241,
4, rue Alfred Kastler, 44307 Nantes, France.
`thierry.petit@mines-nantes.fr`, `narendra.jussien@mines-nantes.fr`

Abstract. This article 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. We experiment our technique with a coarse-grained algorithm for Max-RPC, called Max-RPC^{rm}, and a variant of it, L-Max-RPC^{rm}. Experiments confirm the interest of strong consistencies for Constraint Programming tools.

1 Introduction

This paper presents a generic framework for integrating strong local consistencies into Constraint Programming (CP) tools, especially event-based solvers. It is successfully experimented using Max-RPC^{rm} and L-Max-RPC^{rm}, recent coarse-grained algorithms for Max-RPC and a variant of this consistencyÂ [25].

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 [23]. We call them event-based solvers. Each propagator is called according to the events that occur in the 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 fixed point is obtained by 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 [9, 6] 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,[4] and they are not used for solving real-life problems.

This article 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* [7, 1, 19], 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 for our global constraint: it is used to apply a propagation technique on a given subset of constraints, as it was done in [20] in the context of over-constrained problems. Our scheme may be connected to BessiÃ¨re & RÃ©gin's "on the fly" subproblem solving [5]. However, there is a fundamental divergence as our scheme is aimed at encoding strong consistencies. Thus, we keep a local evaluation of the supports for each constraint in the encapsulated model.

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. This feature is an alternative to the heuristics for dynamically switching between different levels of consistency during search [21]. 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 encapsulated constraint. Event-based solvers generally do not provide such a level of precision.

We experiment our framework with the Max-RPC strong consistency [8], using the Choco CP solver [15]. We use a coarse-grained algorithm for Max-RPC, called Max-RPC^{rm} [25]. This algorithm exploits backtrack-stable data structures in a similar way to AC-3^{rm} [17].

Section 2 presents the background about constraint networks and local consistencies useful to understand our contributions. Section 3 presents the generic integration scheme and it specialization to specific strong local consistencies. Section 4 describes Max-RPC^{rm} and L-Max-RPC^{rm}. Section 5 details the ex-

---

[4] Some strong consistencies such as SAC [3] can be implemented using assignment and propagation methods, and some solvers may feature such ones.

perimental evaluation of our work. Finally, we discuss the perspectives and we conclude.

2 Background

A *constraint network* $\mathcal{N}$ is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ which consists of :

- a set of n variables $\mathcal{X}$,
- a set of domains $\mathcal{D}$, where the domain $\mathrm{dom}(X) \in \mathcal{D}$ of the variable X is the finite set of at most d values that the variable X can take, and
- a set $\mathcal{C}$ of e constraints that specify the allowed combinations of values for given subsets of variables.

A variable/value couple (X, v) will be denoted X_v. An *instantiation* I is a set of variable/values couples. I is *valid* iff for any variable X involved in I, $v \in \mathrm{dom}(X)$. 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 $(\mathrm{vars}(C), \mathrm{rel}(C))$, where $\mathrm{vars}(C)$ is a set of k variables and $\mathrm{rel}(C)$ is a relation of arity k. $I[X]$ denotes the value of X in the instantiation I. $C_{XY\ldots Z}$ denotes a constraint such that $\mathrm{vars}(C) = \{X, Y, \ldots, Z\}$. Given a constraint C, an instantiation I of $\mathrm{vars}(C)$ (or of a superset of $\mathrm{vars}(C)$, considering only the variables in $\mathrm{vars}(C)$), *satisfies* C iff $I \in \mathrm{rel}(C)$. We say that I is *allowed* by C.

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

2.1 Local consistencies

Definition 1 (Support). *Let C be a constraint and $X \in \mathrm{vars}(C)$. A* **support** *for a value $a \in \mathrm{dom}(X)$ w.r.t. C is an instantiation $I \in \mathrm{rel}(C)$ such that $I[X] = a$.*

Definition 2 (Arc-consistency). *Let C be a constraint and $X \in \mathrm{vars}(C)$. Value $a \in \mathrm{dom}(X)$ is* **arc-consistent** *w.r.t. C iff it has a support in C. C is arc-consistent iff $\forall X \in \mathrm{vars}(C)$, $\mathrm{dom}(X)$ is arc-consistent.*

$\mathcal{N}(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is arc-consistent iff $\forall X \in \mathcal{X}, \forall a \in \mathrm{dom}(X), \forall C \in \mathcal{C}, a$ is arc-consistent w.r.t. C.

Definition 3 (Closure). *Let $\mathcal{N}(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, Φ a local consistency (e.g., AC) and $\mathcal{C}$ a set of constraints $\subseteq \mathcal{C}$. $\Phi(\mathcal{D}, \mathcal{C})$ is the* closure *of $\mathcal{D}$ for Φ on $\mathcal{C}$, i.e. the set of domains obtained from $\mathcal{D}$ 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 [9, 6] 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 paper focuses on domain filtering consistencies [9], which only prune values from domains and leave the structure of the constraint network unchanged.

Binary constraint networks. Firstly, w.r.t. binary constraint networks, as it is mentioned in [6], (i, j)-consistency [11] 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 a $(1, 1)$-consistency.

A binary constraint network $\mathcal{N}$ that has non empty domains is :

Path Consistent (PC) iff it is $(2, 1)$-consistent.

Path Inverse Consistent (PIC) [12] iff it is $(1, 2)$-consistent.

Restricted Path Consistent (RPC) [2] 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.

Max-Restricted Path Consistent (Max-RPC) [8] iff it is $(1, 1)$-consistent and for each value X_a, and each variable $Y \in \mathcal{X} \backslash X$, one consistent extension Y_b of X_a is $(2, 1)$-consistent (that is, can be extended to any third variable).

Singleton Arc-Consistent (SAC) [3] iff each value is SAC, and a value X_a is SAC if the 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).

Non-binary constraint networks. Concerning non binary constraint networks, relational arc- and (i, j)-consistencies [10] provide the concepts useful to extend local consistencies defined for binary constraint networks to the non-binary case. A constraint network $\mathcal{N}$ that has non empty domains is:

Relational AC (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.

Relational (i, j)-consistent iff any consistent instantiation for i of the variables in a set of j constraints can be extended to all the variables in the set.

From these notions, new domain filtering consistencies for non-binary constraints inspired by the definitions of RPC, PIC and Max-RPC were proposed in [6]. Moreover, some interesting results were obtained using pairwise consistency. A constraint network $\mathcal{N}$ that has non empty domains is :

Pairwise Consistent (PWC) [14] iff it has no empty relations and any locally consistent instanciation from the relation of a constraint can be consistently extended to any other constraint that intersects with this. One may apply both PWC and GAC.
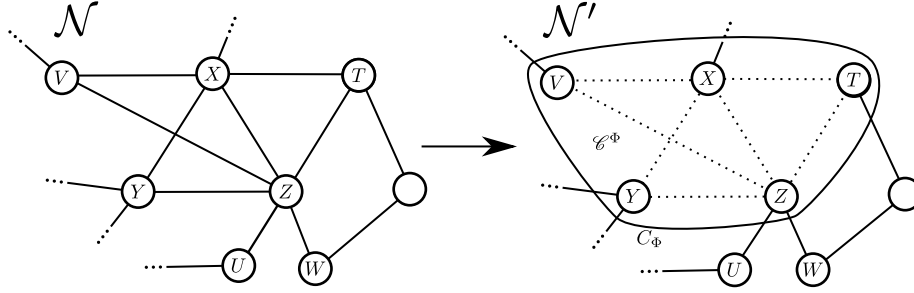
Fig. 1. A strong consistency global constraint C_Φ, used to enforce the strong local consistency on a subset of constraints $\mathscr{C}^\Phi$. $\mathcal{N}'$ is the new network obtained when replacing $\mathscr{C}^\Phi$ by the global constraint.

Pairwise Inverse Consistent (PWIC) [22] iff for each value X_a, there is a support for a w.r.t. all constraints involving X, such that 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, 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.
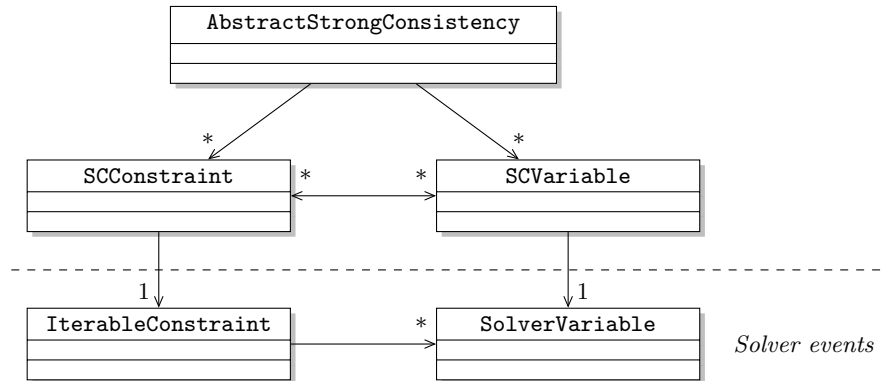
Fig. 2. UML Class diagram [13] of the integration of strong local consistencies into event-based solvers. Arrows describe association relations with cardinalities, either one (1) or many (*).

Note that the original constraints of the problem can be kept in place, so that they can perform their standard pruning task before the stronger consistency is applied. For best efficiency, however, the solver should feature constraint prioritization (see *e.g.*, [26]): propagating the weaker constraints after the strong consistency constraint would be useless.

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 `IterableConstraint`. 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 (relational) (i, j)-consistencies (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 and export these, as it is performed to handle logical connectives in [18]. 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.

To give access to and iterate over the supports, the methods `firstSupport` and `nextSupport` are specified in `IterableConstraint`, a subclass of the abstract constraint class of the solver.

Generic iterators. The `firstSupport` and `nextSupport` services are not usually available in most constraint solvers. However, a generic implementation can be
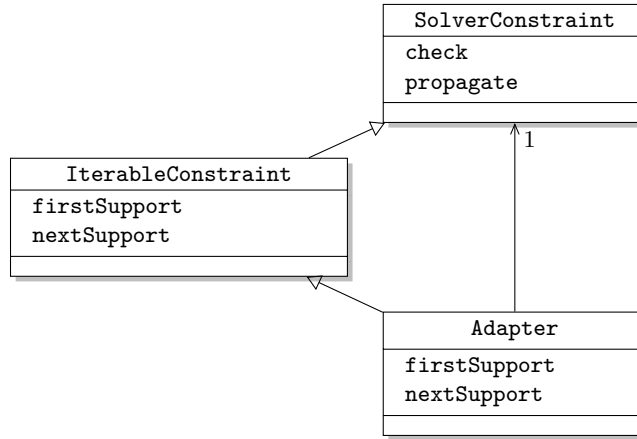
Fig. 3. A generic implementation of support iterator functions, given the constraints provided by a solver. Following the UML specifications, open triangle arrows describe generalization relations.

devised, either by relying on *constraint checkers*[5] (all valid instantiations are checked until an allowed one is found), or by using directly the propagator of the constraint. To perform this, one can simply build a search tree which enumerates the solutions to the CSP composed of the constraint and the variables it involves. These implementations are wrapped in an `Adapter` class that specializes the required `IterableConstraint` 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.

Specialized iterators. For some constraints, more efficient, ad-hoc algorithms for `firstSupport` and `nextSupport` functions can be provided (*e.g.*, for positive table constraints [4]). As `IterableConstraint` specializes `SolverConstraint` (see Figure 3), it is sufficient to specialize `IterableConstraint` for this purpose.

Some strong consistencies such as Path Consistency may be implemented by directly using the propagators of the constraints [16]. 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 domain and remove values. Class `SCVariable` is used for representing the constraint subnetwork $(\text{vars}(C_\Phi), \mathcal{D}, \mathcal{C}^\Phi)$. A link is kept with the solver variable for operation on domains.

---

[5] A constraint checker checks whether a given instantiation is allowed by the constraint or not.
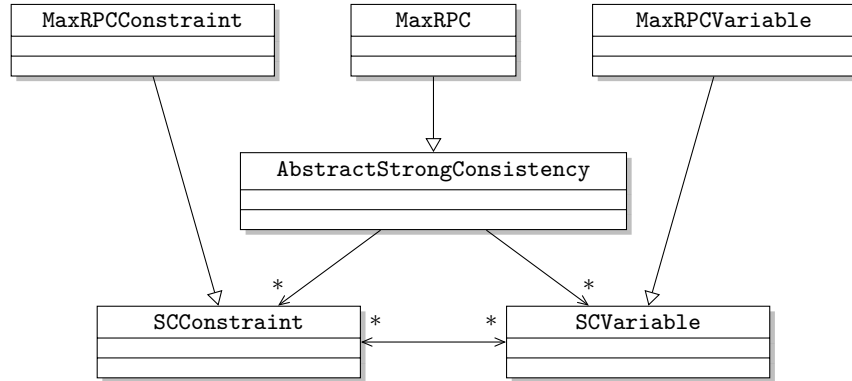
Fig. 4. Diagram of the integration of Max-RPC into event-based solvers.

The main feature of `SCVariable` is to "hide" the external constraints from the point of view of the `AbstractStrongConsistency` class implementations. Moreover, it may prove to be very useful to specialize the `SCVariable` class to add data structures required by the strong consistency implementation.

Variable degree-based heuristics. Some 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 [8]. 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 specialized to efficiently manipulate 3-cliques.

4   A coarse grained algorithm for Max-RPC

This section presents the implementation of Max-RPC we used in section 5 to experiment our approach.

Algorithm 1: MaxRPC$(P = (\mathscr{X}, \mathscr{C}), \mathscr{Y})$

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

1 $\mathscr{Q} \leftarrow \mathscr{Y}$;
2 **while** $\mathscr{Q} \neq \emptyset$ **do**
3 $\quad$ pick X from $\mathscr{Q}$;
4 $\quad$ **foreach** $Y \in \mathscr{X} \mid \exists C_{XY} \in \mathscr{C}$ **do**
5 $\quad\quad$ **foreach** $v \in \text{dom}(Y)$ **do** **if** $\texttt{revise}(C_{XY}, Y_v, \textbf{true})$ **then** $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Y\}$;
6 $\quad$ **foreach** $(Y, Z) \in \mathscr{X}^2 \mid \exists (C_{XY}, C_{YZ}, C_{XZ}) \in \mathscr{C}^3$ **do**
7 $\quad\quad$ **foreach** $v \in \text{dom}(Y)$ **do** **if** $\texttt{revisePC}(C_{YZ}, Y_v, X)$ **then** $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Y\}$;
8 $\quad\quad$ **foreach** $v \in \text{dom}(Z)$ **do** **if** $\texttt{revisePC}(C_{YZ}, Z_v, X)$ **then** $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Z\}$;

Algorithm 2: revisePC(C_{YZ}, Y_a, X): boolean

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

1 **if** $pcRes[C_{YZ}, Y_a][X] \in \text{dom}(X)$ **then return false** ;
2 $b \leftarrow \texttt{findPCSupport}(Y_a, Z_{last[C_{YZ}, Y_a]}, X)$;
3 **if** $b = \bot$ **then return** $\texttt{revise}(C_{YZ}, Y_a, \textbf{false})$;
4 $pcRes[C_{YZ}, Y_a][X] \leftarrow b$; **return false**;

Max-RPC^{rm} [25] is a *coarse-grained* algorithm for Max-RPC. This algorithm exploits backtrack-stable data structures inspired from AC-3^{rm} [17]. *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 [17] that AC-3^{rm} behaves better than the optimal algorithm in most cases. In [25], authors demonstrate that using a coarse-grained approach is also especially interesting for the strong local consistency Max-RPC. With g being the maximal number of constraints involving a single variable, c the number of 3-cliques and s the maximal number of 3-cliques related to the same constraint ($s < g < n$ and $e \leq ng/2$), the worst-case time complexity for Max-RPC^{rm} is $O(eg + ed^3 + csd^4)$ and its space complexity is $O(ed + cd)$.

L-Max-RPC^{rm} is a variant of Max-RPC^{rm} that computes a relaxation of Max-RPC with a worst-case time complexity in $O(eg + ed^3 + cd^4)$ and a space complexity in $O(c + ed)$ (that is, a space complexity very close to best AC algorithms). The pruning performed by L-Max-RPC^{rm} is strictly stronger than that of AC.

Algorithms 1 to 4 describe Max-RPC^{rm} and L-Max-RPC^{rm}. In this algorithm, Lines 6-8 of Algorithm 1 and Lines of 5-8 of Algorithm 3 are added to a standard AC-3^{rm} algorithm. L-Max-RPC^{rm} removes the memory and time overhead caused by the *pcRes* data structure and the calls to the $\texttt{revisePC}$ function.

Algorithm 3: 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*$\land res[C_{XY}, Y_a] \in \mathrm{dom}(X)$ **then return false** ;
2 $b \leftarrow$ firstSupport$(C_{XY}, \{Y_a\})[X]$;
3   **while** $b \neq \bot$ **do**
4 $PConsistent \leftarrow$ **true** ;
5   **foreach** $Z \in \mathscr{X} \mid (X, Y, Z)$ *form a 3-clique* **do**
6 $c \leftarrow$ findPCSupport(Y_a, X_b, Z) ;
7   **if** $c = \bot$ **then** $PConsistent \leftarrow$ **false** ; **break**;
8 $currentPcRes[Z] \leftarrow c$;
9   **if** $PConsistent$ **then**
10 $res[C_{XY}, Y_a] \leftarrow b$; $res[C_{XY}, X_b] \leftarrow a$;
11 $pcRes[C_{XY}, Y_a] \leftarrow pcRes[C_{XY}, X_b] \leftarrow currentPcRes$;
12   **return false** ;
13 $b \leftarrow$ nextSupport$(C_{XY}, \{Y_a\}, \{X_b, Y_a\})[X]$;
14 remove a from $\mathrm{dom}(Y)$; **return true** ;

**Algorithm 4**: findPCSupport(X_a, Y_b, Z): value

---

1 $c_1 \leftarrow$ firstSupport$(C_{XZ}, \{X_a\})[Z]$;
2 $c_2 \leftarrow$ firstSupport$(C_{YZ}, \{Y_b\})[Z]$;
3   **while** $c_1 \neq \bot \land c_2 \neq \bot \land c_1 \neq c_2$ **do**
4   **if** $c_1 < c_2$ **then**
5 $c_1 \leftarrow$ nextSupport$(C_{XZ}, \{X_a\}, \{X_a, Z_{c_2-1}\})[Z]$;
6   **else**
7 $c_2 \leftarrow$ nextSupport$(C_{YZ}, \{Y_b\}, \{Y_b, Z_{c_1-1}\})[Z]$;
8   **if** $c_1 = c_2$ **then return** c_1 ;
9   **return** $\bot$;

The principle is to modify Algorithm 1 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 3 (greyed parts in the algorithms). 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 L-Max-RPC^{rm} will be filtered. The consistency enforced by L-Max-RPC^{rm} in not monotonous and will depend on the order in which the modified variables are picked from $\mathscr{Q}$, but its filtering power is only slightly weaker than that of Max-RPC on random problems, despite the significant gains in space and time complexities.
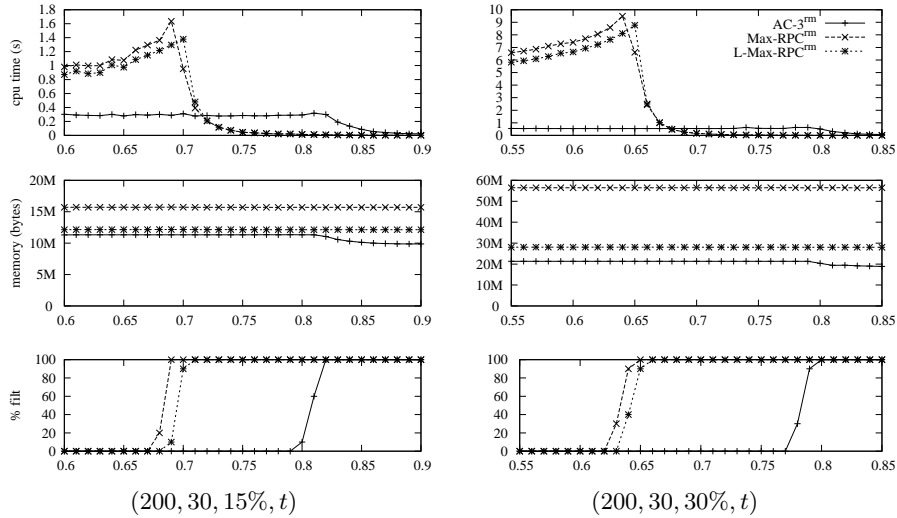
Fig. 5. Initial propagation: CPU time, memory and % of removed values against tightness on homogeneous random problems (200 variables, 30 values, 15/30% density).

5 Experiments

The aim of our experiments is to show the practicability of our approach. We evaluate (1.) the eventual overload of the integration, and (2.) the interest of mixing various consistencies, as is made possible thanks to our scheme.

We implemented the diagram of Figure 4 in Choco [15], using the algorithm for Max-RPC described in section 4. In our experiments, Max-RPC^{rm} and L-Max-RPC^{rm} are compared to Choco's native AC-3^{rm} filtering algorithm.

5.1 Evaluating the overload

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[6] of the constraint graph and the tightness[7] of the constraints.

Single propagation. Figure 5 compares the time and memory used for the initial propagation on rather large problems (200 variables, 30 values), as well as the percentage of removed values. In our experiments, only constraints that form a 3-clique are mapped to the global constraint. A low density leads to a

---

[6] The density is the proportion of constraints in the graph w.r.t. the maximal number of possible constraints, i.e. $\gamma = e / \binom{n}{2}$.

[7] The tightness is the proportion of instantiations forbidden by each constraint.
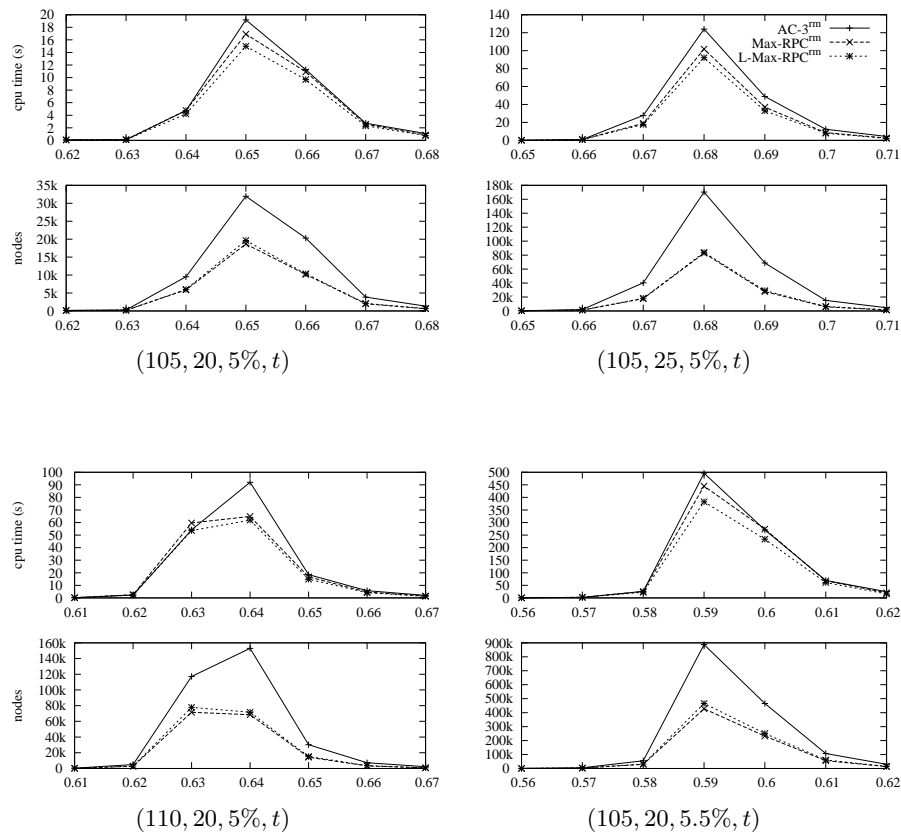
Fig. 6. Full search: cpu time and nodes against tightness on homogeneous random problems (105-110 variables, 20-25 values).

low number of 3-cliques, hence experimental results are coherent with theoretical complexities.

Full search. Figure 6 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 [8, 25] showed that maintaining Max-RPC in a dedicated solver was interesting for large and sparse problems, compared with maintaining AC. Our results show that encoding Max-RPC within a global constraint leads to the same conclusions, hence that our scheme has no incidence on computation costs.

		AC-3^{rm}	L-Max-RPC^{rm}	AC-3^{rm}+L-Max-RPC^{rm}
$(35, 17, 44\%, 31\%)$	*cpu (s)*	**6.1**	11.6	non
	nodes	21.4k	8.6k	applicable
$(105, 20, 5\%, 65\%)$	*cpu (s)*	20.0	**16.9**	non
	nodes	38.4 k	19.8 k	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	96.8	103.2	**85.1**
$+(105, 20, 5\%, 65\%)$	*nodes*	200.9k	107.2k	173.4k
$(110, 20, 5\%, 64\%)$	*cpu (s)*	73.0	**54.7**	non
	nodes	126.3k	56.6k	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	408.0	272.6	**259.1**
$+(110, 20, 5\%, 64\%)$	*nodes*	773.0k	272.6k	316.5k

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

5.2 Mixing local consistencies

A new feature provided by our approach is the ability to mix various levels of local consistency for solving a given constraint network, each on some *a priori* disjoint subsets of constraints.[8]

Table 1 shows the effectiveness of the new possibility of mixing two levels of consistency within the same model. The first two row corresponds to the median results over 50 instances of problems $(35, 17, 44\%, 31\%)$ and $(105, 20, 5\%, 65\%)$. The first problem is better resolved by using AC-3^{rm} while the second one shows better results with L-Max-RPC^{rm}.

The 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 L-Max-RPC 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.

Our results show that 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 paper 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 soundness of this feature is validated by our experiments. A major

---

[8] Such constraints can share variables.

interest of our schema is that strong consistencies can be applied with different kinds of constraints, including user-defined constraints.

Although our contribution is not restricted to event-based solvers, we underline that an important motivation for providing this scheme was to bridge the gap between strong consistencies and event-based constraint toolkits. Such toolkits put together many scientific contributions of the community. They provide users with advanced APIs that allow to use a catalog of global constraints with powerful filtering algorithms, to implement new constraints, to define specific search strategies, to hybrid CP with other solving techniques such as Local Search (*e.g.*, Comet [24]), or to integrate explanations (*e.g.*, Choco [15]). Our approach adds to this list of features the use of strong consistencies.

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. This second perspective may allow to link our approach with the heuristics for adapting the level of consistency during the search process [21].

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. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, SICS, 2005.
2. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings of IEEE-CAIA'95*, 1995.
3. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1):29–41, 2008.
4. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 1997.
5. C. Bessière and J.-C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *CP*, pages 103–117, 1999.
6. C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.
7. C. Bessière and P. van Hentenryck. To be or not to be... a global constraint. In *Proceedings CP'03*, pages 789–794. Springer, 2003.
8. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings of CP'97*, pages 312–326, 1997.
9. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
10. R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.
11. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
12. E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *AAAI/IAAI, Vol. 1*, pages 202–208, 1996.

13. Object Management Group. Unified Modeling Language (UML). `http://www.omg.org/spec/UML/`, 2000–2010.

14. P. Janssen, P. Jegou, 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.

15. F. Laburthe, N. Jussien, et al. Choco: An open source Java constraint programming library. `http://choco.emn.fr/`, 2008.

16. C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proceedings of CP'2007*, 2007.

17. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.

18. O. Lhomme. Arc-Consistency Filtering Algorithms for Logical Combinations of Constraints. In *Proceedings of CPAIOR'04*, pages 209–224, 2004.

19. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

20. 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.

21. K. Stergiou. Heuristics for dynamically adapting propagation. In *ECAI*, pages 485–489, 2008.

22. K. Stergiou and T. Walsh. Inverse consistencies for non-binary constraints. *Proceedings of ECAI*, 6:153–157, 2006.

23. P. van Hentenryck, Y. Deville, and CM. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

24. P. van Hentenryck, L. Michel, A. See, et al. The Comet Programming Language and System. `http://www.comet-online.org`, 2001–2007.

25. J. Vion and R. Debruyne. Light Algorithms for Maintaining Max-RPC During Search. In *Proceedings of SARA'09*, 2009.

26. J. Vion and S. Piechowiak. Handling Heterogeneous Constraints in Revision Ordering Heuristics. In *Proc. of the TRICS'2010 workshop held in conjunction with CP'2010*, 2010.