A Generic Customizable Framework for Inverse Local Consistency

Gérard Verfaillie and **David Martinez**
ONERA-CERT
2 av. Edouard Belin, BP 4025
31055 Toulouse Cedex 4, France
{verfaillie,martinez}@cert.fr

Christian Bessière
LIRMM-CNRS (UMR 5506)
161 rue Ada
34392 Montpellier Cedex 5, France
bessiere@lirmm.fr

Abstract

Local consistency enforcing is at the core of *CSP* (*Constraint Satisfaction Problem*) solving. Although *arc consistency* is still the most widely used level of local consistency, researchers are going on investigating more powerful levels, such as *path consistency*, *k-consistency*, *(i,j)-consistency*. Recently, more attention has been turned to *inverse local consistency* levels, such as *path inverse consistency*, *k-inverse consistency*, *neighborhood inverse consistency*, which do not suffer from the drawbacks of the other local consistency levels (changes in the constraint definitions and in the constraint graph, prohibitive memory requirements).
In this paper, we propose a *generic framework* for inverse local consistency, which includes most of the previously defined levels and allows a rich set of new levels to be defined. The first benefit of such a generic framework is to allow a user to define and test many different inverse local consistency levels, in accordance with the problem or even the instance he/she has to solve. The second benefit is to allow a *generic algorithm* to be defined. This algorithm, which is parameterized by the chosen inverse local consistency level, generalizes the *AC7* algorithm used for *arc consistency*, and produces from any instance its locally consistent closure at the chosen level.

Motivations

Local consistency enforcing techniques are at the core of the *CSP* (*Constraint Satisfaction Problem* (Mackworth 1992)) solving techniques and are probably the main ingredients of their success. Thanks to local reasoning, they allow any instance P to be simplified, by eliminating values or combinations of values that cannot be involved in any solution of P. More precisely, from any instance P, they produce a new instance $\widehat{P}$, with the following properties:

(1) when compared with P, $\widehat{P}$ is simplified: some values have been removed from the variable domains of P, some tuples of values have been removed from the constraint relations of P;

(2) P and $\widehat{P}$ are equivalent: they have the same set of solutions;

(3) $\widehat{P}$ is locally consistent at the specified level.

$\widehat{P}$ is often referred to as the locally consistent closure of P at the specified level of consistency. Always thanks to local reasoning, local consistency enforcing may allow global inconsistency to be detected, when a variable domain or a constraint relation becomes empty.

Used in a preprocessing step, they allow either a search for a solution to be started with a simplified instance, or sometimes any search to be avoided in case of inconsistency detection. Used at each node of a search tree, they allow branches that do not lead to any solution to be cut earlier. Lastly, used in the framework of an interactive solving, they allow consequences of the user's choices (variable assignment, variable domain reduction, constraint adding) to be explicited.

After this picture, it may seem strange that the most widely used local consistency level is one of the simplest: *arc consistency* whose enforcing removes from the domain of a variable v the values that cannot be consistently extended to another variable v'. Although various extensions of *arc consistency* have been proposed (*path consistency*, *k-consistency*, *(i,j)-consistency* (Freuder 1978; 1985)), it remains that they are not often used. There are three reasons for that:

(a) a worst case time complexity of the associated algorithms, which grows quickly (for example, exponentially as a function of k with *k-consistency* (Cooper 1989));

(b) a worst case space complexity, which grows the same way (for example, also exponentially as a function of k with *k-consistency*);

(c) the recording of forbidden tuples, which implies either to create new extensionally defined constraints (changes in the constraint network), or to add an extensional definition to existing possibly intensionally defined constraints (changes in the constraint definitions).

Starting from these observations, more attention has been recently turned to inverse local consistency levels, such as *path inverse consistency*, *k-inverse consistency*, *neighborhood inverse consistency*, *restricted path consistency*, *max restricted path consistency* or *singleton arc consistency* (Freuder & Elfe 1996; Debruyne & Bessière 1997a), which

do not suffer from all the drawbacks of the previous local consistency levels.

Informally speaking, for all these levels but the last three, inverse local consistency enforcing removes from the domain of a variable v the values that cannot be consistently extended to some additional variables. *Arc consistency*, whose enforcing removes values that cannot be consistently extended to any other variable, is the simplest level of inverse local consistency. *Path inverse consistency* enforcing removes values that cannot be consistently extended to any set of two other variables. *k-inverse consistency* enforcing removes values that cannot be consistently extended to any set of *k-1* other variables. *Neighborhood inverse consistency* enforcing removes values that cannot be consistently extended to the set of variables directly linked to v.

Since inverse local consistency enforcing never removes combinations of values, and then does not create new constraints and does not modify existing constraint definitions, it does not suffer from the drawbacks (b) and (c) listed above[1]:

(b') the memory that is necessary for recording the removed values is $O(nd)$, if n is the number of variables and d the maximum size of the variable domains;

(c') there is no change, neither in the constraint graph, nor in the constraint definitions.

It can be observed that most of the defined inverse local consistency levels consider the ability to extend the assignment of a variable v with a value val to some sub-instances, involving v and some additional variables, and that they *differ from each other only in the definition of the considered sub-instances*. This observation paves the way for a generic definition of inverse local consistency. In this paper, we propose such a definition. Its advantages are multiple:

- it allows most of the inverse local consistency levels, separately and differently presented in the literature, to be brought together;
- doing that, it makes their theoretical comparison easier, for example, in terms of filtering power and worst case complexity of the associated algorithms;
- it paves the way for the definition of many new levels, either generic, or specifically defined for an instance or a kind of instance;
- it allows a generic algorithm for inverse local consistency enforcing, parameterized by the chosen level, to be defined; in this paper, we propose an algorithm that generalizes the *AC7* algorithm used for *arc consistency*;
- doing that, it saves a lot of time that otherwise would be wasted, for each new level, in algorithm definition, implementation and debugging tasks;
- it allows a user to define and test easily many levels of inverse local consistency and to build the level that is the most suited to the instance or kind of instance he/she has to solve.

[1]Note that the drawback (a) does not disappear: as we will see in the next section, the worst case time complexity of the associated algorithms grows exponentially as a function of the number of additional variables.

A generic framework for inverse local consistency

A generic definition of inverse local consistency

The generic definition of inverse local consistency we propose is naturally based on the notion of *viability* of a value in a sub-instance:

Definition 1 *Let $P = (V, C)$ be an instance, defined by a set V of variables (a finite domain $d(v)$ being associated with each variable $v \in V$) and a set C of constraints. A sub-instance of P is an instance $P' = (V', C')$, where $C' \subseteq C$ and V' is the set of the variables in V that are linked by the constraints in C'.*

Definition 2 *Let $P = (V, C)$ be an instance and $P' = (V', C')$ be a sub-instance of P. Let $v \in V'$ and $val \in d(v)$. The value (v, val) is said to be viable in P' iff the sub-instance P' restricted by the assignment $v \leftarrow val$ is consistent,* i.e. *has at least one solution.*

If a value (v, val) is not viable in a sub-instance P', then it is not involved in any solution of P', and thus not involved in any solution of P. It can be removed from the domain of v without losing any solution.

We can easily extend this notion of viability to variables and *(sub-instance, set of variables)* pairs:

Definition 3 *Let $P = (V, C)$ be an instance and $P' = (V', C')$ be a sub-instance of P. Let $v \in V'$. The variable v is said to be viable in P' iff all the values in $d(v)$ are viable in P'. Let $V'' \subseteq V'$. The pair (P', V'') is said to be viable iff P' is consistent and all the variables in V'' are viable in P'.*

In the latter definition, the condition enforcing the consistency of P' may seem useless. It has been added to take into account pairs (P', V'') where $V'' = \varnothing$.

The idea is to define an inverse local consistency level ilc by a function $\mathbf{def}_{ilc}$, which associates with any instance P a set $\mathbf{def}_{ilc}(P)$ of *(sub-instance, set of variables)* pairs whose viability has to hold. Then, we can define the inverse local consistency of an instance P at the level ilc (denoted as *ilc-consistency*) as follows:

Definition 4 *Let $P = (V, C)$ be an instance. Let ilc be a local consistency level and $\mathbf{def}_{ilc}$ be the associated function. P is said to be ilc-consistent iff all the pairs of* (sub-instance, set of variables) *in $\mathbf{def}_{ilc}(P)$ are viable.*

According to this definition, specifying a function $\mathbf{def}_{ilc}$ is sufficient to specify an inverse local consistency level ilc. In the next subsection, we define the function $\mathbf{def}$ associated with some well known inverse local consistency levels. In the subsection after, we point out how this definition can be used to specify new inverse local consistency levels, either generic or specific.

Already known inverse local consistency levels

In the general framework of binary or non-binary CSPs, the functions **def** associated with *arc consistency*, *neighborhood inverse consistency* and *global consistency* can be defined as follows:

- For each constraint $c \in C$, *arc consistency* (ac) (Mackworth 1977) considers the sub-instance P' involving the set $v(c)$ of variables linked by c and the constraint c itself; it requires that each variable in $v(c)$ be viable in P'; then, $\mathbf{def}_{ac}((V,C)) = \{((v(c),\{c\}),v(c))/c \in C\}$;
- For each variable $v \in V$, *neighborhood inverse consistency* (nic) (Freuder & Elfe 1996) considers the sub-instance P' involving the set $lv(v)$ of variables directly linked to v and the set $clv(v)$ of constraints linking these variables[2]; it requires that v be viable in P'; then, $\mathbf{def}_{nic}((V,C)) = \{((lv(v),clv(v))),\{v\})/v \in V\}$;
- *Global consistency* (gc) (Freuder 1991; Dechter 1992)[3] is the highest inverse local consistency level; it requires that each variable in V be viable in P, then, $\mathbf{def}_{gc}((V,C)) = \{((V,C),V)\}$.

Other known inverse local consistency levels like, for example, *path inverse consistency* or *k-inverse consistency* (Freuder & Elfe 1996) can be easily expressed in this framework.

New inverse local consistency levels

In addition to these levels, the framework we defined paves the way for the definition of multiple new inverse local consistency levels, as limitless as the number of functions **def** we can imagine. Here are some examples:

- *k-length neighborhood inverse consistency* (k-$lnic$) is a generalization of nic, which considers, for each variable v, the sub-instance involving all the variables that are linked to v using a path whose length is less than or equal to k; for example, nic is equivalent to 1-$lnic$;
- *k-neighborhood inverse consistency* (k-nic) is a restriction of nic, which considers only sub-instances involving a number of variables less than or equal to k; there are several ways to enforce this limitation; we can consider only the variables whose number of linked variables is less than or equal to k; we can also consider all the variables, but, for each variable, consider only k linked variables (or several sets of k linked variables), chosen according to any criterion: domain size, arity and tightness of the involved constraints, etc
- all the previously defined inverse local consistency levels (in this subsection and in the previous one) can be restricted by considering only the constraints whose tightness is greater than or equal to a given threshold or whose arity is smaller than or equal to another threshold; as in the example of k-nic, they can also be restricted by considering only sub-instances whose size (number of variables and domain sizes) is less than or equal to a given threshold.

Using prior knowledge on the structure of the instance or of the kind of instance he/she has to solve, the user can define and test suited inverse local consistency levels. To do that, he/she only needs to define the associated functions **def**. Note that specifying functions **def** such that $\forall (P',V'') \in \mathbf{def}(P), V'' = \varnothing$ allows him/her to define local consistency levels that enforce only sub-instance consistency, without any value removal.

A generic algorithm for inverse local consistency enforcing

General description

The generic algorithm we propose is an extension of the *AC7* algorithm (Bessière, Freuder, & Régin 1999), which is currently considered as one of the most efficient algorithms for *arc consistency* enforcing. It is widely drawn from (Bessière & Régin 1997) and (Bessière & Régin 1998). As in (Bessière, Freuder, & Régin 1999), it is based on the notion of *supporting assignment*:

Definition 5 *Let $P = (V,C)$ be an instance and $P' = (V',C')$ be a sub-instance of P. Let $v \in V'$ and $val \in d(v)$. The assignment SA of the variables in V' is a supporting assignment for the value (v,val) in the sub-instance P' iff $SA[v] = val$* [4] *and SA is solution of P'.*

According to Definitions 2 and 5, a value is viable in a sub-instance iff it has a supporting assignment in this sub-instance. If not, it is not viable and can be removed, without losing any solution.

As a result, for each triple (v,val,P') it has to check, the algorithm searches for a supporting assignment of (v,val) in P'. If such an assignment SA is found, (v,val) is recorded as supported by SA and for all the values $(v',val') \in SA$, SA is recorded as supported by (v',val') in P'. If such an assignment is not found, the value val is removed from the domain of v and all the values supported by an assignment that is itself supported by (v,val) in any sub-instance P', have to search for a new supporting assignment in P'.

Data structure

The data structure we use has six main components:

(CL) a list CL (for *Checking List*) of *(sub-instance, set of variables)* pairs (P',V'') whose viability has to be checked; this list is initialized by a call to the function $\mathbf{def}_{ilc}$ associated with the considered inverse local consistency level;

(SIL) for each variable v, a static list $SIL(v)$ (for *Sub Instance List*) of sub-instances $P' = (V',C')$ that have to be checked again when a value val is removed from the domain of v;

[2] Two variables are said to be directly linked iff they are involved in the same constraint. Given a set V' of variables, a constraint c is said to link these variables iff $v(c) \subseteq V'$.

[3] (Freuder 1991) speaks of *completable* values and (Dechter 1992) of *globally consistent* variables. If n is the number of variables, one refers also to *global consistency* as *(1,n-1)-consistency* or *n-inverse consistency*.

[4] $SA[v]$ denotes the value of the variable v in the assignment SA.

(SSL) a list SSL (for *Seeking Support List*) of *(variable, value, sub-instance)* triples (v, val, P'), associated with the values (v, val) that are searching for a supporting assignment in the sub-instance P';

(SVL) for each supporting assignment SA, a list $SVL(SA)$ (for *Supported Value List*) of values that are supported by SA;

(SAL) for each triple (v, val, P'), a list $SAL(v, val, P')$ (for *Supported Assignment List*) of supporting assignments that are themselves supported by (v, val) in P';

(MSA) for each triple (v, val, P'), an assignment $MSA(v, val, P')$ (for *Minimum-Supporting Assignment*), such that no supporting assignment exists before $MSA(v, val, P')$ for (v, val) in P', according to the lexicographic ordering of the assignments of P' defined by the static orderings chosen for variables and values;

Figure 1 gives a graphical representation of the supporting links between values and assignments and between assignments and values (data structures SAL and SVL).

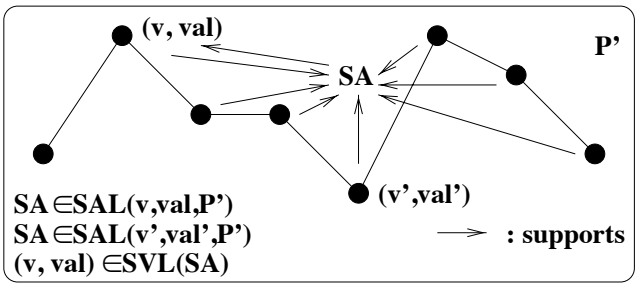

Figure 1: Graphical representation of the supporting links.

Algorithm

The high level pseudo-code of the generic algorithm is shown in Figures 2 to 7.

The principal novelty with respect to $AC7$ lies in the use of the function **search** called by the function **seek-next-support**. This function searches for a new supporting assignment of a value (v, val) in a sub-instance P'. The method used for searching for such an assignment depends on the size of the sub-instance:

- if P' contains only one constraint, as with ac, the search is a simple enumeration;
- if it contains more than one constraint, as with nic, k-nic, k-$lnic$, gc or any other inverse local consistency level, a tree search is performed.

Any of the techniques that have been developed for *CSP* solving can be selected for this search. The SAL structure avoids searching if a supporting assignment can be directly inferred: if a value supports an assignment, this value is also supported by this assignment. The MSA structure avoids exploring parts of the search space that have been previously explored, but imposes static variable and value

```
function local-consistency-enforcing(P, ilc)
CL ← init-cl(P, ilc);
while CL ≠ ∅
      (P', V") ← pick-and-remove(CL);
      if V" = ∅
      then if not consistent(P')
            then return false;
      else SSL ← init-ssl(P', V");
            if not seek-supports(SSL, ilc)
            then return false;
return true;
```

Figure 2: Inverse local consistency enforcing of an instance P at the level ilc.

```
function seek-supports(SSL, ilc)
while SSL ≠ ∅
      (v, val, P') ← pick-and-remove(SSL);
      if not removed(val, domain(v))
      then if not seek-inferred-support(v, val, P')
            then if not seek-next-support
                        (v, val, P', MSA(v, val, P'))
                  then remove(val, domain(v));
                        if domain(v) = ∅
                        then return false;
                        SSL ← add-ssl(SSL, v, val);
return true;
```

Figure 3: Searching for supporting assignments for a list SSL of *(variable, value, sub-instance)* triples.

```
function seek-inferred-support(v, val, P')
if SAL(v, val, P') ≠ ∅
then SA ← pick-not-remove(SAL(v, val, P'));
      add((v, val), SVL(SA));
      return true;
return false;
```

Figure 4: Searching for an inferred supporting assignment for a value (v, val) in a sub-instance P'.

```
function seek-next-support(v, val, P', MSA)
SA ← search(v, val, P', MSA);
if SA ≠ ∅
then forall (v', val') ∈ SA
            add(SA, SAL(v', val', P'));
      SVL(SA) ← {(v, val)};
      MSA(v, val, P') ← SA;
      return true;
return false;
```

Figure 5: Searching for a supporting assignment for a value (v, val) in a sub-instance P', starting from MSA.

```
function init-cl(P, ilc)
init-sil(P)
CL ← def_ilc(P);
init-sil-sal (CL);
return CL;

function init-sil(P)
forall v ∈ variables(P)
        SIL(v) ← ∅;

function init-sil-sal(CL)
forall (P',V") ∈ CL
        forall v ∈ variables(P')
                add(P', SIL(v));
                forall val ∈ d(v)
                        SAL(v, val, P') ← ∅;

function init-ssl(P',V")
SSL ← ∅;
forall v ∈ V"
        forall val ∈ d(v)
                SSL ← add((v, val, P'), SSL);
                MSA(v, val, P') ← ∅;
return SSL;
```

Figure 6: Initialization of the data structures.

orderings. If one wants to use dynamic orderings or, more generally, if one wants to derive more benefit from the previous searches for a supporting assignment of any value in the same sub-instance, any of the techniques of solution or reasoning reuse that have been developed for *dynamic CSPs* (Verfaillie & Schiex 1994b; Schiex & Verfaillie 1994; Verfaillie & Schiex 1994a), can be considered. Moreover, any level of inverse local consistency can be maintained during this search: *forward checking*, *arc consistency*, etc. Our current implementation (Martinez 1998) uses *forward checking* and static variable and value orderings.

It is easy to show informally that such an algorithm turns any instance P into an instance $\widehat{P}$, which is simplified, equivalent and locally consistent at the specified level:

(1) $\widehat{P}$ is simplified because some values involved in P have been removed;

(2) it is equivalent because all the instances successively generated during algorithm execution are equivalent to P: let $\widehat{P}_c$ be the current instance resulting from the removal of some values; initially, $\widehat{P}_c = P$; let us assume that, at a given step of the execution, $\widehat{P}_c$ is equivalent to P; if a value is removed, it is not viable in a sub-instance $\widehat{P}'_c$ of $\widehat{P}_c$, and thus not involved in any solution of $\widehat{P}_c$; the resulting new current instance is consequently equivalent to the previous one, and thus equivalent to P;

(3) it is locally consistent at the specified level, because, at the end of the algorithm, all the remaining values are viable in all the associated sub-instances.

```
function add-ssl(SSL, v, val)
forall P' ∈ SIL(v)
        forall SA ∈ SAL(v, val,P')
                forall (v', val') ∈ SA
                        remove(SA, SAL(v', val', P'));
                forall (v",val") ∈ SVL(SA)
                        add((v",val",P'), SSL);
                remove(SA);
```

Figure 7: Update of the data structure SSL.

Its time and space complexity can be easily obtained by generalizing the reasoning used for *AC7*. Let p be the number of *(sub-instance, set of variables)* pairs $(P' = (V', C'), V'')$ to consider, n' be the maximum number of variables in V', n'' be the maximum number of variables in V'', and d be the maximum domain size. Time and space complexities are respectively $O(p.dn''.d^{n'-1})$ and $O(p.dn''.n')$.

Extending definitions and algorithms

As explained in the previous section, as soon as the number of constraints involved in the sub-instance to consider is greater than 1, a tree search is performed in order to prove the viability or the non-viability of a value. But other less expensive methods could be considered.

First of all, any limited local search can be used to establish viability; if it succeeds, viability is proven. If not, a systematic tree search is needed.

Second, inverse local consistency enforcing can be used to prove non-viability. This implies an extension of the notions of viability and inverse local consistency previously introduced. Definitions 2 and 4 can be replaced by the following recursive definitions:

Definition 6 *Let $P = (V, C)$ be an instance and $P' = (V', C')$ be a sub-instance of P. Let ilc_2 be an inverse local consistency level. Let $v \in V'$ and $val \in d(v)$. The value (v, val) is said to be ilc_2-viable in P' iff the sub-instance P' restricted by the assignment $v \leftarrow val$ is ilc_2-consistent.*

Definition 7 *Let $P = (V, C)$ be an instance. Let ilc_1 and ilc_2 be two inverse local consistency levels and $\mathbf{def}_{ilc_1}$ be the function associated with ilc_1. P is said to be ilc_1-ilc_2-consistent iff all the pairs in $\mathbf{def}_{ilc_1}(P)$ are ilc_2-viable.*

Note that recursivity stops when ilc_2 equals *consistency*. This extension allows other levels, previously known or not, to be included. For example:

- *singleton arc consistency* (Debruyne & Bessière 1997b) could be defined as *gc-ac* ($ilc_1 = gc$, $ilc_2 = ac$); *singleton neighborhood inverse consistency* could be defined as *gc-nic*; more generally, *singleton ilc* could be defined as *gc-ilc*;

- nic-ac would be an interesting trade-off between *neighborhood inverse consistency* and *arc consistency*: for each variable v, one considers the sub-instance P' involving all the variables that are directed linked to v and all the constraints that link these variables; but for each value val of v, only *arc consistency* of P' restricted by the assignment $v \leftarrow val$ is required.

Conclusion and perspectives

As a conclusion, we have defined a generic customizable framework for inverse local consistency, which includes most of the previously defined levels and allows as many new levels as we can imagine to be built, according to the instance or kind of instance we have to solve. Let us point out that this framework does not include specific inverse local consistency levels, such as *restricted path consistency* and *max restricted path consistency*: their inclusion would have implied important changes in the proposed framework and a loss in terms of simplicity and clarity.

Associated with this framework, we have defined a generic algorithm, parameterized by the chosen level. This algorithm has been implemented in the frame of an interactive tool for CSP solving. No comparison in terms of efficiency has been carried out between this generic algorithm and other algorithms dedicated to a given level. Genericity may induce a loss of efficiency. But one can argue that a generic carefully implemented algorithm is often more efficient than a specific rapidly implemented algorithm.

Always associated with this framework, it would be worth developing a small language, allowing the user to express easily any level he/she wants. Finally, two remarks:

- the current constraint programming tools do not specify the levels of local consistency they use; "opening the box" and offering at least an advanced user the means of defining this level may allow him/her, on the one hand, to understand better how local consistency enforcing works and, on the other hand, to tune it according to the instance or kind of instance he/she has to solve;
- the generic framework we defined reverses the "landscape" of CSP solving: whereas local consistency enforcing is most of the time presented as a subroutine for tree search, tree search is presented here as a subroutine for local consistency enforcing; but nothing prevents us from defining a tree search, which calls local consistency enforcing, which in turn calls another tree search, etc.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and Bertrand Cabon, Simon de Givry, Michel Lemaitre, Lionel Lobjois and Thomas Schiex for stimulating discussions about this work.

References

Bessière, C., and Régin, J. 1997. Arc Consistency for General Constraint Networks: Preliminary Results. In *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 398–404.

Bessière, C., and Régin, J. 1998. Local Consistency on Conjunctions of Constraints. In *Proc. of the ECAI-98 Workshop on "Non-Binary Constraints"*, 53–59.

Bessière, C.; Freuder, E.; and Régin, J. 1999. Using Inference to Reduce Arc Consistency Computation. *Artificial Intelligence* 107:125–148.

Cooper, M. 1989. An Optimal k-Consistency algorithm. *Artificial Intelligence* 41:89–95.

Debruyne, R., and Bessière, C. 1997a. From Restricted Path Consistency to Max-Restricted Path Consistency. In *Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming (CP-97)*, 312–326.

Debruyne, R., and Bessière, C. 1997b. Some Practical Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 412–417.

Dechter, R. 1992. From Local to Global Consistency. *Artificial Intelligence* 55:87–107.

Freuder, E., and Elfe, C. 1996. Neighborhood Inverse Consistency Preprocessing. In *Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 202–208.

Freuder, E. 1978. Synthesizing Constraint Expressions. *Communications of the ACM* 21(11):958–966.

Freuder, E. 1985. A Sufficient Condition for Backtrack-Bounded Search. *Journal of the ACM* 32(4):755–761.

Freuder, E. 1991. Completable Representations of Constraint Satisfaction Problems. In *Proc. of the 1st International Conference on the Principles of Knowledge Representation and Reasoning (KR-91)*, 186–195.

Mackworth, A. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8(1):99–118.

Mackworth, A. 1992. Constraint Satisfaction. In Shapiro, S., ed., *Encyclopedia of Artificial Intelligence*. John Wiley & Sons. 285–293.

Martinez, D. 1998. *Résolution Interactive de Problèmes de Satisfaction de Contraintes*. Thèse de doctorat, ENSAE, Toulouse, France.

Schiex, T., and Verfaillie, G. 1994. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools* 3(2):187–207.

Verfaillie, G., and Schiex, T. 1994a. Dynamic Backtracking for Dynamic Constraint Satisfaction Problems. In *Proc. of the ECAI-94 Workshop on "Constraint Satisfaction Issues Raised by Practical Applications"*.

Verfaillie, G., and Schiex, T. 1994b. Solution Reuse in Dynamic Constraint Satisfaction Problems. In *Proc. of the 12th National Conference on Artificial Intelligence (AAAI-94)*, 307–312.