# Failed Value Consistencies for Constraint Satisfaction

Christophe Lecoutre, Olivier Roussel

# Failed Value Consistencies for Constraint Satisfaction

Christophe Lecoutre and Olivier Roussel

Univ Lille Nord de France, F-59000 Lille, France
UArtois, CRIL, F-62307 Lens, France
CNRS, UMR 8188, F-62307 Lens, France
{lecoutre,roussel}@cril.fr

**Abstract.** In constraint satisfaction, basic inferences rely on some properties of constraint networks, called consistencies, that allow the identification of inconsistent instantiations (also called nogoods). Two main families of consistencies have been introduced so far: those that permit us to reason from variables such as $(i, j)$-consistency and those that permit us to reason from constraints such as relational $(i, j)$-consistency. This paper introduces a new family of consistencies based on the concept of failed value (a value pruned during search). This family is orthogonal to previous ones.

## 1 Introduction

Any user of a constraint solver ideally expects the system to be robust and clever enough to automatically identify all relevant properties of a problem instance. These properties typically depend on the structure of the instance and can help solving it. Some of the approaches to reach that goal are inferences from strong consistencies, adaptive heuristics, nogood recording and automatic symmetry breaking. They permit an efficient exploration of the search space, learning much useful information before or during search so as to avoid exploring fruitless combinations of values of variables.

Usually, backtrack search is used for solving instances of the Constraint Satisfaction Problem (CSP). Backtrack search combines a depth-first exploration to instantiate variables and a backtracking mechanism to deal with encountered dead-ends. During search, some values are proved to be inconsistent, i.e. not to participate to any solution - we call them failed values. Interestingly, it is known [13] that failed values "convey" some information: given a satisfiable binary CSP instance $P$, for any pair $(x, a)$ where $x$ is a variable of $P$ and $a$ a value in the domain of $x$, if there is no solution containing $a$ for $x$, then there is necessarily a variable $y \neq x$ which is assigned a value $b$ such that $(y, b)$ is not compatible with $(x, a)$. It is then possible to dynamically and iteratively decompose problem instances [13, 2].

In this paper, we propose to exploit failed values in a different manner. By locally reasoning from failed values, we show that some inferences can be performed within reasonable complexities. In particular, we develop a new family of domain-filtering consistencies based on failed values and show that they are complementary to (incomparable with) usual ones. They contribute to prune the search space and also offer a lazy detection of a generalized form of the substitutability relation. Algorithms checking or enforcing consistencies based on failed values can be naturally grafted to any constraint propagation engine in order to reinforce the filtering strength of the search algorithm. This approach may represent an interesting contribution to the quest for robust solvers.

After recalling some technical background, two basic consistencies, called FVC and AFVC and based on failed values, are presented. Next, algorithms for checking FVC and enforcing AFVC are introduced, and AFVC is compared to substitutability and usual consistencies. Finally, before presenting some preliminary experimental results, we show that an entire class of consistencies based on failed values can be naturally defined.

## 2 Technical Background

A constraint network (CN) $P$ is composed of a finite set of $n$ variables, denoted by $vars(P)$, and a finite set of $e$ constraints, denoted by $cons(P)$. Each variable $x$ has an associated domain, denoted by $dom(x)$, that contains the finite set of values that can be assigned to $x$. Each constraint $c$ involves a set of variables, called the *scope* of $c$ and denoted by $scp(c)$. It is defined by a relation, denoted by $rel(c)$, which contains the set of tuples allowed for the variables involved in $c$. A *binary* constraint involves exactly 2 variables, and a *non-binary constraint* strictly more than 2 variables. For a binary constraint $c_{xy}$ such that $scp(c_{xy}) = \{x, y\}$, if $(a, b) \in rel(c_{xy})$, we say that $(x, a)$ and $(y, b)$ are compatible. We also say that $(x, a)$ and $(y, b)$ are compatible if there is no binary constraint between $x$ and $y$.

In this paper, we shall consider given an initial CN $P^{init}$ and a current CN $P$ derived from $P^{init}$ by potentially reducing variable domains. The initial domain of a variable $x$ is denoted by $dom^{init}(x)$ whereas the current domain is denoted by $dom^P(x)$ or more simply by $dom(x)$. A (current) value of $P$ is a pair $(x, a)$ with $x \in vars(P)$ and $a \in dom(x)$. For any variable $x$, we always have $dom(x) \subseteq dom^{init}(x)$ and denote this fact by $P \preceq P^{init}$. More generally, given two CNs $P$ and $P'$, we note $P' \preceq P$ iff $P$ and $P'$ are defined on the same set of variables and the same set of constraints, and for every variable $x$ in $vars(P) = vars(P')$, we have $dom^{P'}(x) \subseteq dom^P(x)$. $P' \prec P$ iff $P' \preceq P$ and there exists a variable $x$ such that $dom^{P'}(x) \subset dom^P(x)$.

An instantiation $I$ of a set $X = \{x_1, \dots, x_k\}$ of variables is a set $\{(x_1, a_1), \dots, (x_k, a_k)\}$ such that $\forall i, a_i \in dom^{init}(x_i)$ ; the set $X$ of variables occurring in $I$ is denoted by $vars(I)$ and each value $a_i$ is denoted by $I[x_i]$. An instantiation $I$ on a CN $P$ is an instantiation of a set $X \subseteq vars(P)$ ; it is complete if $vars(I) = vars(P)$, partial otherwise. $I$ is valid on $P$ iff $\forall (x, a) \in I, a \in dom(x)$ $(= dom^P(x))$. $I[x/a]$ is the instantiation obtained from $I$ by replacing the value assigned to $x$ in $I$ by $a$. An instantiation $I$ covers a constraint $c$ iff $scp(c) \subseteq vars(I)$, and satisfies a constraint $c$ with $scp(c) = \{x_1, \dots, x_r\}$ iff a) $I$ covers $c$ and b) the tuple $(a_1, \dots, a_r)$, such that $\forall i, a_i = I[x_i]$, is allowed by $c$, i.e. $(a_1, \dots, a_r) \in rel(c)$. An instantiation $I$ on a CN $P$ is locally consistent iff a) $I$ is valid on $P$ and b) every constraint of $P$ covered by $I$ is satisfied by $I$. It is locally inconsistent otherwise. A solution of $P$ is a complete instantiation on $P$ that is locally consistent. An instantiation $I$ on a CN $P$ is globally inconsistent, or a nogood, iff it cannot be extended to a solution of $P$. It is globally consistent otherwise.

A CN is said to be *satisfiable* iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable or not. A CSP instance is defined by a CN which is solved either by finding a solution or by proving unsatisfiability. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Typically, constraint propagation algorithms are based on some properties of CNs that allow us to identify and remove some values which cannot occur in any solution. Such properties are called domain-filtering consistencies [6, 4].

Let us introduce some classical consistencies. An instantiation $I$ on $P$ is a support (resp. a conflict) for a value $(x, a)$ on a constraint $c$ involving $x$ iff $I$ is valid, $I[x] = a$ and $I$ satisfies (resp. does not satisfy) $c$. A value $(x, a)$ of $P$ is GAC-consistent (GAC stands for Generalized Arc Consistency) iff there exists a support for $(x, a)$ on every constraint of $P$ involving $x$. $P$ is GAC-consistent iff every value of $P$ is GAC-consistent. For binary CNs, GAC is referred to as AC (Arc Consistency). For any CN $P$, we know that there exists a greatest GAC-consistent network, denoted by $GAC(P)$ and called the GAC-closure of $P$, which is equivalent to $P$ and such that $GAC(P) \preceq P$. When the domain of a variable of $P$ is empty, $P$ is clearly unsatisfiable, which is denoted by $P = \bot$. The CN $P|_{x=a}$ is obtained from $P$ by removing every value $b \neq a$ from $dom(x)$. A value $(x, a)$ of $P$ is SAC-consistent (SAC stands for Singleton Arc Consistent) iff $GAC(P|_{x=a}) \neq \bot$. $P$ is SAC-consistent iff every value of $P$ is SAC-consistent.

Any domain-filtering consistency allows us to identify and remove inconsistent values. In order to compare the pruning capability of different consistencies, we can introduce a preorder [6]. Let $\phi$ and $\psi$ be two consistencies. $\phi$ is stronger than $\psi$, denoted by $\phi \trianglerighteq \psi$, iff whenever $\phi$ holds on a CN $P$, $\psi$ also holds on $P$. $\phi$ is strictly stronger than $\psi$, denoted by $\phi \triangleright \psi$ iff $\phi \trianglerighteq \psi$ and there exists at least one CN $P$ such that $\psi$ holds on $P$ but not $\phi$. When some consistencies cannot be ordered (none is stronger that the other), we say that they are *incomparable*. For classical domain-filtering consistencies defined on binary CNs, we have: SAC $\triangleright$ MaxRPC $\triangleright$ PIC $\triangleright$ AC (MaxRPC and PIC are respectively Max-Restricted Path Consistency and Path Inverse Consistency, see [3]).

## 3 Consistencies based on Failed Values

In this section, we present two new basic consistencies. The first identifies nogoods of any size whereas the second identifies inconsistent values (i.e. nogoods of size 1). These two consistencies are based on *failed values* which are simply values proved to be inconsistent (e.g. during search). Failed values "convey" some information:

**Lemma 1 (directly derived from [13]).** *If a value $(x, a)$ of a CN $P$ is globally inconsistent then every solution $S$ of $P$ is such that $S[x/a]$ violates at least one constraint of $P$ involving $x$.*

*Proof.* $S[x/a]$ is not a solution of $P$ since $(x, a)$ is globally inconsistent. This means that at least one constraint of $P$ is not satisfied by $S[x/a]$. But we know that every constraint $c$ of $P$ that does not involve $x$ is satisfied by $S[x/a]$ because the restriction of $S[x/a]$ over $scp(c)$ is exactly the restriction of $S$ over $scp(c)$. Consequently, at least one constraint of $P$ involving $x$ is not satisfied by $S[x/a]$. □

If $P$ is a binary CN, then every solution of $P$ contains a value for a variable $y \neq x$ which is not compatible with $(x, a)$. A failed value is defined as follows:

**Definition 1.** *Let $P$ and $P'$ be two CNs such that $P' \prec P$. A* failed value *of $P'$ with respect to $P$ is a value $(x, a)$ of $P$ such that $P|_{x=a}$ is unsatisfiable and $a \notin dom^{P'}(x)$ .*

In practice, a failed value is a value pruned from a CN because it has been proved to be inconsistent. At any time during search, a failed value can be identified by inference and/or search methods [7, 16]. For example, if $P^{init}|_{x=a}$ is shown to be unsatisfiable, clearly, $a$ can be removed from $dom^{init}(x)$. We then obtain a smaller CN $P$ with $(x, a)$ being a failed value of $P$ (with respect to $P^{init}$). However, note that failed values can be defined with respect to any intermediate CN reached during search. We need now to introduce *conflict sets*.

**Definition 2.** *Let $P$ be a CN, $x$ be a variable of $P$ and $a \in dom^{init}(x)$.*

- *The* conflict set *of $(x, a)$ on a constraint $c$ of $P$ involving $x$, denoted by $\chi_P(c, x, a)$, is the set of valid instantiations $I$ of $scp(c) \setminus \{x\}$ on $P$ such that $I \cup \{(x, a)\}$ does not satisfy $c$.*
- *The* conflict set *of $(x, a)$ on $P$ is $\chi_P(x, a) = \cup_{c \in cons(P) | x \in scp(c)} \chi_P(c, x, a)$.*

For every conflict set $\chi$, $vars(\chi) = \cup_{I \in \chi} vars(I)$. When possible, we simplify $\chi_P(x, a)$ into $\chi(x, a)$. Figure 1 shows[1] a simple CN $P$ with a binary constraint between $w$ and $x$ and a ternary constraint between $w, y$ and $z$. Here $\chi(w, a) = \{\{(x, b)\}, \{(y, a), (z, a)\}\}$ and $\chi(w, c) = \{\{(x, b)\}, \{(x, c)\}, \{(y, c), (z, c)\}\}$; note that $\{(x, b)\}$ and $\{(y, a), (z, a)\}$ are two instantiations in $\chi(w, a)$ of size 1 and 2.

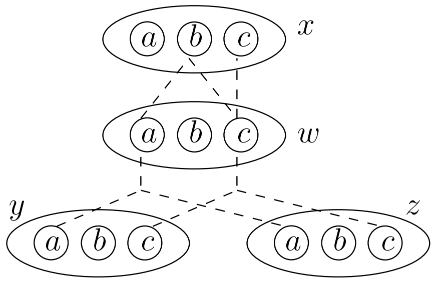

**Fig. 1.** Illustration of conflict sets.

Failed values and instantiations can be connected as follows:

[1] In each figure of this paper, solid (resp. dashed) edges represent allowed (resp. forbidden) tuples. The absence of edges between two variables means that there is no binary constraint involving them.

**Definition 3.** *Let $(x, a)$ be a failed value of a CN $P$ and $I$ a valid instantiation on $P$.*

- $(x, a)$ *is* covered *by $I$ iff* $vars(\chi_P(x, a)) \subseteq vars(I)$.
- $(x, a)$ *is* verified *by $I$ iff* $\exists J \in \chi_P(x, a) \mid J \subseteq I$.

Note that a failed value verified by an instantiation is not necessarily covered by it. However, it is shown below that when a failed value is covered by an instantiation but not verified, a nogood is identified. FVC (Failed Value Consistency) is a general nogood-identifying consistency.

**Definition 4.** *Let $P$ be a constraint network.*

- *A valid instantiation $I$ on $P$ is* FVC-consistent *for a failed value $(x, a)$ of $P$ iff either $(x, a)$ is not covered by $I$ or $(x, a)$ is verified by $I$.*
- *A valid instantiation $I$ on $P$ is* FVC-consistent *iff it is FVC-consistent for every failed value of $P$; otherwise, $I$ is said to be* FVC-inconsistent*.*

Assume that $(w, c)$ in Figure 1 is a failed value. $I = \{(x, a), (y, c), (z, c)\}$ is an instantiation that verifies $(w, c)$ because $I$ contains $\{(y, c), (z, c)\} \in \chi(w, c)$. $I' = \{(x, a)\}$ does not verify $(w, c)$ but is FVC-consistent for $(w, c)$ because $(w, c)$ is not covered by $I'$. $I'' = \{(x, a), (y, a), (z, a)\}$ is FVC-inconsistent because $(w, c)$ is both covered by $I''$ and not verified by $I''$.

**Proposition 1.** *Any FVC-inconsistent instantiation is globally inconsistent.*

*Proof.* Without any loss of generality, we consider here that $I$ is a valid instantiation on a constraint network $P$ that is FVC-inconsistent for a failed value $(x, a)$ of $P$ with respect to $P^{init}$; we have $P \prec P^{init}$. We know that there is no solution of $P^{init}$ involving $(x, a)$ because $(x, a)$ is a failed value of $P$ wrt $P^{init}$. We can even say more: for every solution $S$ of $P^{init}$, the complete instantiation $S[x/a]$ is not a solution because at least one constraint involving $x$ is violated (see Lemma 1). Because $I$ is FVC-inconsistent for $(x, a)$, we know that $I$ covers $vars(\chi_P(x, a))$ and $(x, a)$ is not verified by $I$. This means that it is not possible to extend $I$ into a complete instantiation $I'$ on $P$ such that $I'[x/a]$ violates at least one constraint involving $x$. Every solution of $P$ is a solution of $P^{init}$ (since $P \prec P^{init}$) and every solution $S$ of $P^{init}$ is such that $S[x/a]$ violates at least one constraint involving $x$. We can deduce that $I$ is globally inconsistent. □

Otherwise stated, some nogoods can be identified via deleted values (that are themselves nogoods). These nogoods are not necessarily of size 1. For example, in Figure 2 there is a failed value $(w, a)$ and three binary constraints involving $w$. Any valid instantiation of $\{x, y, z\}$ is globally inconsistent if it only contains values compatible with $(w, a)$. In other words, every tuple in $C_x \times C_y \times C_z$ is a nogood (of size 3).

Interestingly, inference can be conducted differently by reasoning between each value and each failed value (through its conflict set). More precisely, we can define a related domain-filtering consistency, called Arc Failed Value Consistency (AFVC).

**Definition 5.** *Let $P$ be a constraint network.*

- *A value $(x, a)$ of $P$ is AFVC-consistent for a failed value $(y, b)$ of $P$ iff $(x, a)$ can be extended to a locally consistent instantiation verifying $(y, b)$.*

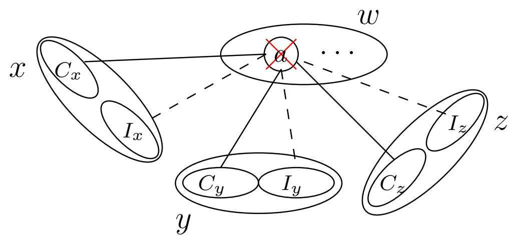

**Fig. 2.** A failed value $(w, a)$, its compatible values in $C_x$, $C_y$ and $C_z$ and its incompatibles values in $I_x$, $I_y$ and $I_z$.

- *A value $(x, a)$ of $P$ is AFVC-consistent iff $(x, a)$ is AFVC-consistent for every failed value of $P$; otherwise, $(x, a)$ is said to be AFVC-inconsistent.*
- *$P$ is AFVC-consistent iff every value of $P$ is AFVC-consistent.*

For the first item above, note that we may have $x = y$. In this case, necessarily, we have $a \neq b$ since $(x, a)$ is a current value whereas $(y, b)$ is a pruned one. Note that AFVC can be regarded as a local consistency since it suffices to reason from the conflict set of each failed value. In particular for binary constraints, a value $(x, a)$ is AFVC-consistent for a failed value $(y, b)$ iff $(x, a)$ is compatible with a valid value in $\chi(y, b)$. The AFVC algorithm given in Section 4 is based on this simple observation.

**Proposition 2.** *Any AFVC-inconsistent value is globally inconsistent.*

*Proof.* Let $(x, a)$ be a current value of $P$ that is AFVC-inconsistent for a failed value $(y, b)$ of $P$. Suppose that there exists a solution $S$ of $P$ such that $S[x] = a$. Necessarily, as $(x, a)$ is AFVC-inconsistent for $(y, b)$, $S[y/b]$ cannot violate any constraint involving $y$, and consequently is a solution of $P$. This contradicts the fact that $(y, b)$ is globally inconsistent (since it is a failed value), and consequently our hypothesis. We can deduce that $(x, a)$ is globally inconsistent. □

As an illustration, let us consider the CN (partially) depicted in Figure 3. This CN can be completed (see dots) so that some classical consistencies hold (e.g. arc consistency). We assume here that $(w, a)$ is a failed value and $\chi(w, a) = \{\{(x, a)\}, \{(y, c)\}\}$.

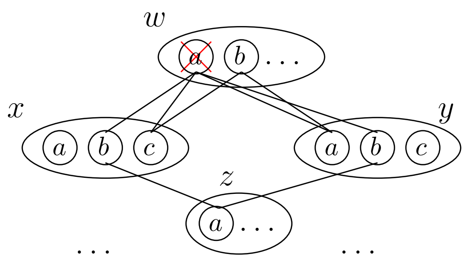

**Fig. 3.** Illustration of AFVC

Observe that $(w, b)$ and $(z, a)$ are AFVC-inconsistent. Indeed, $(w, b)$ (resp. $(z, a)$) is not compatible with any value in $\chi(w, a)$.

Restricted to nogoods of size 1, FVC is strictly weaker than AFVC (the proof is omitted). We have:

**Proposition 3.** *Let $(x, a)$ be a value of $P$. If $I = \{(x, a)\}$ is FVC-inconsistent then $(x, a)$ is AFVC-inconsistent.*

Finally, one can show that AFVC verifies certain properties (e.g. see [1, 3, 16]) that permits us to define the AFVC-closure.

**Proposition 4.** *For any CN $P$, there exists a greatest AFVC-consistent CN equivalent to $P$, called the AFVC-closure of $P$ and denoted by $AFVC(P)$, such that $AFVC(P) \preceq P$.*

$AFVC(P)$ can be obtained by iteratively removing, in any order, values that are not AFVC-consistent.

## 4 Algorithms for FVC and AFVC

We propose to embed filtering algorithms based on failed values within MAC (Maintaining Arc Consistency) [23]. MAC is a backtrack search algorithm that develops a binary search tree: at each node, an uninstantiated variable $x$ is selected, a value $a$ in $dom(x)$ is selected, a left subtree starting with a branch labelled with the positive decision $x = a$ (variable assignment) is first explored and a right subtree starting with a branch labelled with the negative decision $x \neq a$ (value refutation) is later explored. The consistency enforced at each node is GAC.

For binary CNs, an immediate solution to make use of FVC is to post, for each failed value $(x, a)$, a non-binary constraint whose scope is $vars(\chi(x, a))$: its associated relation forbids any instantiation FVC-inconsistent for $(x, a)$. For our example in Figure 2, we would obtain a ternary constraint $c_{xyz}$ such that $rel(c_{xyz}) = dom(x) \times dom(y) \times dom(z) \setminus C_x \times C_y \times C_z$. Interestingly, one may conceive efficient filtering algorithms (propagators) to enforce GAC on such constraints. However, this approach is intrusive and its generalization to non-binary CNs is rather complex. This is why we propose a weakened approach for the general case: checking FVC at each node of the search tree by checking that no identified failed value is currently covered and unverified. To identify failed values during search, it suffices to keep the set $F$ of negative decisions labelling the current branch (i.e. the branch leading from the root of the search tree to the current node). Note that failed values that correspond to values removed when enforcing GAC can be discarded because these failed values are always verified (as long as no domain wipe-out occurs): for each of these values, there exists a constraint with only conflicts.

To check FVC, Algorithm 1 is called. For each failed value, we seek a conflict; seekConflict$(c, x, a)$ seeks a conflict for $(x, a)$ on $c$. When a conflict is found for a failed value $(x, a)$, it is stored in a backtrack-stable data structure called $res$; this plays the role of residues as in [17] when establishing GAC. The validity of the residual conflict is first tested at line 3. When no conflict exists for a failed value, $false$ is returned,

```
function checkFailedValue($P$: CN, $(x, a)$: failed value):Boolean
begin
    if isValid($res[x, a]$) then
        return true
    foreach constraint $c$ of $P$ such that $x \in scp(c)$ do
        $\tau \leftarrow$ seekConflict$(c, x, a)$
        if $\tau \neq nil$ then
            $res[x, a] \leftarrow \tau$
            return true
    return false
end
function checkFVC($P$: CN, $F$: set of failed values):Boolean
begin
    foreach failed value $(x, a) \in F$ do
        if $\neg$checkFailedValue($P$, $(x, a)$) then
            return false
    return true
end
```

**Algorithm 1**: Checking FVC

which forces the search algorithm to backtrack. The worst-case space complexity of this algorithm is $O(nd)$ whereas its worst-case time complexity is $O(|F|ed^{r-1})$ where $|F|$ denotes the number of failed values in $F$, $e$ the number of constraints, $d$ the greatest domain size and $r$ the greatest constraint arity. For binary constraint networks, the worst-case time complexity is only $O(|F|ed)$.

We now propose an algorithm to enforce the domain-filtering consistency AFVC on binary constraint networks. Given a binary CN $P$ with a set of failed values $F$, the procedure enforceAFVC (see Algorithm 2) computes $AFVC(P)$ and returns $false$ when a domain wipe-out occurs. The data structures are as follows. For each failed value $(x, a)$, $\chi(x, a)$ is considered to be an array of values indexed from 1 to $length(\chi(x, a))$. These values correspond to the instantiations (of size 1 since $P$ is binary) in the conflict set of $(x, a)$. The 2-dimensional array $last$ maps each pair composed of a failed value $(x, a)$ in $F$ and a value $(y, b)$ of $P$ to an an integer corresponding to the index of the most recent value found in $\chi(x, a)$ that is present in $P$ and compatible with $(y, b)$: $last[(x, a)][(y, b)]$ indicates the position of the last found (so-called) AFVC-support for $(y, b)$ on $(x, a)$. For each value $(z, c)$ of $P$, $S(z, c)$ is a list storing the pairs (failed value,value) for which $(z, c)$ is the last found AFVC-support. Structures $S$ and $last$ are inspired from those used in AC6 and AC2001 (see e.g. [3]).

Certainly, dynamically computing (or updating) conflict sets at each node would be prohibitive. This is why we consider that conflict sets are computed for the initial problem instance by the call initialize($P^{init}$). The function enforceAFVC tries to identify an AFVC-support for each pair (failed value,value). If no support can be found for a value $(y, b)$ on a failed value $(x, a)$, $b$ is removed from $dom(y)$ and $(y, b)$ is added to the propagation queue $Q$. Each removed value $(z, c) \in Q$ is "propagated": a new sup-

```
procedure initialize($P$: binary CN)
begin
    foreach value $(x,a)$ of $P$ do
        $\chi(x,a) \leftarrow \varnothing$
        $S(x,a) \leftarrow \varnothing$
    foreach constraint $c_{xy}$ of $P$ do
        foreach tuple $(a,b) \in dom(x) \times dom(y) \mid (a,b) \notin rel(c_{xy})$ do
            add $(x,a)$ to $\chi(y,b)$
            add $(y,b)$ to $\chi(x,a)$
end
function seekAFVCSupport($(x,a)$: failed value, $(y,b)$: value): Boolean
begin
    $position \leftarrow last[(x,a)][(y,b)] + 1$
    while $position \leq length(\chi(x,a))$ do
        $(z,c) \leftarrow \chi(x,a)[position]$
        if $c \in dom(z) \wedge (y,b)$ and $(z,c)$ are compatible then
            $last[(x,a)][(y,b)] \leftarrow position$
            add $((x,a),(y,b))$ to $S(z,c)$
            return true
        $position \leftarrow position + 1$
    return false
end
function enforceAFVC($P$: binary CN, $F$: set of failed values): Boolean
begin
    $Q \leftarrow \varnothing$
    foreach failed value $(x,a) \in F$ do
        foreach value $(y,b)$ of $P$ do
            $last[(x,a)][(y,b)] \leftarrow 0$
            if $\neg$seekAFVCSupport($(x,a),(y,b)$) then
                remove $b$ from $dom(y)$
                if $dom(y) = \varnothing$ then
                    return false
                add $(y,b)$ to $Q$
    while $Q \neq \varnothing$ do
        pick and delete $(z,c)$ from $Q$
        foreach $((x,a),(y,b)) \in S(z,c)$ do
            if $b \in dom(y) \wedge \neg$seekAFVCSupport($(x,a),(y,b)$) then
                remove $b$ from $dom(y)$
                if $dom(y) = \varnothing$ then
                    return false
                add $(y,b)$ to $Q$
        $S(z,c) \leftarrow \varnothing$
end
```

**Algorithm 2**: Enforcing AFVC

port must be found for each pair stored in $S(z, c)$ (this is done from the last recorded position).

The worst-case space and time complexities of enforceAFVC are $O(nd(M+|F|))$ and $O(M|F|nd)$ respectively, where $n$ is the number of variables, $d$ the greatest domain size, $|F|$ the number of failed values and $M$ the maximum size of a conflict set ($M = max_{(x,a)\in P^{init}}|\chi(x, a)|$). Indeed, the space required by $\chi$ arrays is $O(ndM)$, and that required by both $last$ and $S$ is $O(|F|nd)$. On the other hand, the cumulated complexity of seekAFVCSupport for each pair (failed value,value) is $O(M)$, and there are $O(|F|nd)$ different pairs. Note that initialize has a $O(ed^2)$ time complexity but this is amortized since initialize is only called initially (for $P^{init}$). By definition, $|F| < nd$ and $M < nd$ hence $M|F|nd < n^3d^3$. In practice, it may be worthwhile to bound the number of failed values and/or to bound the maximum size of conflict sets by a constant in order to concentrate only on promising failed values. If both are bound, the complexity becomes $O(nd)$. The algorithm presented above can be easily adapted to be used at each node of the search tree developed by MAC (the complexity remains the same for a branch of the search tree).

## 5 Substitutability and Usual Consistencies

Neighborhood substitutability is a weak form of substitutability [11] that can be related to consistencies based on failed values. A value $a \in dom(x)$ is neighborhood substitutable for a value $b \in dom(x)$ iff for every constraint $c$ involving $x$ and every support $I$ for $(x, b)$ on $c$, $I[x/a]$ is a support for $(x, a)$ on $c$. For example, it can be exploited as a reduction operator by applying a convergent sequence of neighborhood substitution deletions [5]. We have the following interesting proposition.

**Proposition 5.** *If a value $(x, a)$ is neighborhood substitutable for a value $(x, b)$ on a CN $P' \succ P$, if $(x, a)$ is a failed value of $P$ and if $(x, b)$ is a value of $P$ then $(x, b)$ is AFVC-inconsistent.*

*Proof.* The definition of neighborhood substitutability can be reformulated as: $(x, a)$ is neighborhood substitutable for $(x, b)$ iff $\chi(x, a) \subseteq \chi(x, b)$. If $(x, a)$ is a failed value of $P$, then it is not possible to extend $(x, b)$ into a consistent instantiation that verifies $(x, a)$. $(x, b)$ is then AFVC-inconsistent. $\square$

AFVC can be seen as a lazy dynamic mechanism to detect values that can be substituted (and are globally inconsistent). Importantly, it allows us to identify inconsistent values for which no neighborhood substitutable value exists. Indeed, a value $(x, b)$ is AFVC-inconsistent if the conflict set of $(x, b)$ is included in the conflict set of a failed value. However, whereas only values for the same variable are considered for neighborhood substitutability, AFVC is more general. An illustration is given in Figure 3: $(w, a)$ is substitutable for $(w, b)$ but not for $(z, a)$ since $w \neq z$.

Interestingly, AFVC is incomparable with most of the domain-filtering consistencies. More precisely, it is incomparable with "usual" consistencies, i.e. local consistencies $\phi$ that do not rely on failed values and that verify the four basic properties: a) $\phi$ holds on any CN only involving entailed constraints (a constraint is entailed on $P$ iff it is satisfied by every valid instantiation on its scope), b) $\phi$ holds on any CN iff it holds on each of its connected sub-networks, c) there exist unsatisfiable CNs where $\phi$ holds and d) there exist some CNs where $\phi$ does not hold. For example, (G)AC, SAC, PIC, ... are "usual" but global consistency (defined as: any locally consistent instantiation can be extended to a solution) is not usual.

**Proposition 6.** *AFVC is incomparable with usual consistencies.*

*Proof.* Let us consider a "usual" local consistency $\phi$. Let us consider a (satisfiable) CN $P_1$ that only contains entailed constraints, a CN $P_2$, unsatisfiable but $\phi$-consistent, on a separate set of variables, and the problem $P = P_1 \cup P_2$. Since $P_2$ is unsatisfiable, any value $(x, a)$ of $P_1$ can be identified as a failed value (e.g. after search). We now assume that $(x, a)$ is a failed value of $P_1$. Since $P_1$ contains only entailed constraints, $\chi(x, a) = \varnothing$ and therefore, no instantiation can verify $(x, a)$. Thus, $P$ is not AFVC-consistent. In contrast, $\phi$ holds on $P$ (by hypothesis), hence $\phi \ntrianglerighteq$ AFVC. Besides, AFVC $\ntrianglerighteq$ $\phi$: it suffices to choose a CN $P$ with no failed value such that $\phi$ does not hold. Consequently, AFVC and $\phi$ are incomparable. $\square$

## 6 A Hierarchy of Consistencies

In [10], Freuder introduced the general class of $(i, j)$-consistencies. Informally, a constraint network is $(i, j)$-consistent iff every locally consistent instantiation of a set of $i$ variables can be extended to a locally consistent instantiation involving any $j$ additional variables. Arc consistency, path consistency [20, 19] and path inverse consistency (PIC) [12] all belong to this class since they correspond to $(1, 1)$-consistency, $(2, 1)$-consistency and $(1, 2)$-consistency, respectively. Another important class of consistencies defined in terms of (existing) constraints is that of relational $(i, m)$-consistencies [8]. Informally, a constraint network is relational $(i, m)$-consistent iff for every set $C$ of $m$ constraints and every set $X \subseteq Y$ of $i$ variables, where $Y = \cup_{c \in C} scp(c)$, every locally consistent instantiation of $X$ can be extended to a valid instantiation of $Y$ satisfying each constraint of $C$. Generalized arc consistency and relational path-inverse-consistency [4] respectively correspond to relational $(1, 1)$-consistency and relational $(1, 2)$-consistency.

Here, we propose a new general class of original consistencies, based on the concept of failed value.

**Definition 6 (Failed Value $(i, f)$-consistency).** *$P$ is FV$(i, f)$-consistent iff for every set $X$ of $i$ variables of $P$ and every set $Y$ of $f$ failed values of $P$, every locally consistent instantiation of $X$ can be extended to a locally consistent instantiation verifying each failed value in $Y$.*

From this general definition, many consistencies can be derived: Arc Failed Value Consistency (AFVC) is FV$(1, 1)$-consistency; Path Failed Value Consistency (PFVC) is FV$(2, 1)$-consistency; Path-Inverse Failed Value Consistency (PIFVC) is FV$(1, 2)$-consistency. Inspired from their variable-based counterparts, MaxRPC and SAC, two additional natural consistencies are introduced.

**Definition 7 (MaxFVC).** *A value $(x, a)$ of $P$ is MaxFVC-consistent iff for every failed value $(y, b)$ of $P$, $(x, a)$ can be extended to a locally consistent instantiation $I$ verifying $(y, b)$ such that for every additional failed value $(z, c)$ of $P$, $I$ can be extended to a locally consistent instantiation verifying $(z, c)$.*

**Definition 8 (SAFVC).** *A value $(x, a)$ of $P$ is SAFVC-consistent iff $AFVC(P|_{x=a}) \neq \perp$.*

**Proposition 7.** *PIFVC $\rhd$ AFVC and SAFVC $\rhd$ AFVC.*

*Proof.* From definitions, we directly deduce that PIFVC $\unrhd$ AFVC and SAFVC $\unrhd$ AFVC. To show strictness, let us consider Figure 4 where a CN $P$ (partially depicted) admits two failed values $(w, a)$, and $(x, a)$. Here, we have $\chi(w, a) = \{\{(y, b)\}, \{(z, b)\}\}$ and $\chi(x, a) = \{\{(y, a)\}, \{(z, b)\}\}$. $P$ is AFVC-consistent but neither SAFVC-consistent nor PIFVC-consistent. Indeed, $(z, a)$ cannot be extended to an instantiation verifying both failed values and $AFVC(P|_{z=a}) = \perp$. □

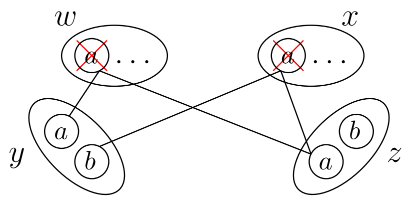

**Fig. 4.** AFVC holds but neither SAFVC nor PIFVC holds.

Figure 5 summarizes the relations between domain-filtering consistencies based on failed values. Due to lack of space, proofs are omitted.

$$\begin{tikzcd}
 & \text{MaxFVC} \arrow[dl] \arrow[dr, leftrightarrow, "\times"] & \\
\text{PIFVC} \arrow[rr, leftrightarrow, "\times"] \arrow[dr] & & \text{SAFVC} \arrow[dl] \\
 & \text{AFVC} &
\end{tikzcd}$$

$\longrightarrow$ strictly stronger

$\overset{\times}{\longleftrightarrow}$ incomparable

**Fig. 5.** Relationships between domain-filtering consistencies based on failed values.

## 7 Preliminary Experimental Results

In order to show the practical interest of consistencies based on failed values, we have performed experiments with our constraint solver Abscon. On a computer equipped with a 3GHz processor and 2GB of RAM under Linux, MAC was employed with $dom/ddeg$ and $lexico$ as variable[2] and value ordering heuristics, as our baseline. We first compared, on binary instances, MAC with MAC embedding the procedure (see Algorithm 1) that checks FVC at each search step (denoted by MAC+FVC) and also with MAC embedding a function that enforces AFVC at each search step (denoted by MAC+AFVC). For our preliminary tests, we have implemented a less sophisticated version of Algorithm 2. This algorithm does enforce AFVC but is quite simpler (and theoretically less efficient) as it does not integrate $last$ and $S$ structures.

**Table 1.** Impact of checking FVC and enforcing AFVC on binary instances.

| | MAC | | MAC+FVC | | MAC+AFVC | |
|---|---|---|---|---|---|---|
| Instance | CPU | nodes | CPU | nodes | CPU | nodes |
| Graph coloring | | | | | | |
| 1-fullins-4-4 | 106 | $7M934$ | 4.8 | $215,812$ | 5.6 | $110,636$ |
| 2-fullins-4-4 | 10.7 | $177,424$ | 4.8 | $67,924$ | 2.3 | $12,764$ |
| 2-insertions-4-3 | 10.5 | $455,533$ | 3.2 | $112,886$ | 5.4 | $62,753$ |
| 2-insertions-5-3 | 3.1 | $7,767$ | 1.8 | $3,494$ | 2.4 | $2,941$ |
| Composed | | | | | | |
| composed-25-5 | 178 | $10M864$ | 85 | $4M838$ | 148 | $2M333$ |
| composed-25-7 | 106 | $7M934$ | 4.78 | $215,812$ | 5.6 | $110,636$ |
| Job-shop | | | | | | |
| os-taillard-4-100-1 | 84 | $1M870$ | 30.2 | $247,383$ | 83 | $206,845$ |
| os-taillard-4-100-3 | 7.6 | $147,270$ | 2.3 | $29,698$ | 3.0 | $17,818$ |
| os-taillard-4-95-2 | 7.5 | $195,665$ | 3.3 | $51,957$ | 19.7 | $41,226$ |
| Queen Attacking | | | | | | |
| qa-5 | 7.5 | $318,601$ | 6.4 | $240,495$ | 10.0 | $238,940$ |
| qa-6 | 311 | $7M703$ | 259 | $5M883$ | 443 | $5M576$ |

Table 1 shows the results obtained on some binary instances[3] with these three backtrack search variants in terms of CPU time and number of visited nodes. Because $dom/ddeg$ is a non-adaptive heuristic, we have the guarantee that MAC+FVC visits less nodes than MAC and that MAC+AFVC visits less nodes than MAC+FVC. In Table 1, for some instances, the number of visited nodes is significantly reduced when consistencies based on failed values are used. MAC+FVC is clearly the most efficient algorithm since it is usually better than MAC+AFVC and sometimes one order of magnitude faster than MAC alone. Interestingly, reasoning from failed values allows us to

[2] Using $dom/wdeg$ does not guarantee exploring the same search tree when additional filtering is performed. This is why we have chosen $dom/ddeg$.

[3] Available at `http://www.cril.fr/~lecoutre/research/benchmarks/`

benefit from the structure (related to substitutability for graph coloring and job-shop instances) contained in these instances. On some other series of instances (not presented here) including random ones, MAC+FVC does not prune the search tree. However, we have observed that this is never penalizing in terms of CPU time. This is because the worst-case time complexity of checking FVC is limited and the number of failed values is usually small. Although one may be disappointed by the relative inefficiency of MAC+AFVC, one should consider that a non optimized AFVC algorithm has been used here and that many developments to control the complexity of AFVC (or even stronger consistencies) remain to be studied.

Table 2 shows the results obtained on some series of non-binary instances. Clearly, MAC+FVC outperforms MAC alone. Finally, note that MAC+FVC was used in Abscon during the third constraint solver competition[4] with good results.

**Table 2.** Impact of checking FVC on non-binary instances.

| | MAC | | MAC+FVC | |
|---|---|---|---|---|
| Instance | CPU | nodes | CPU | nodes |
| Dimacs | | | | |
| hole-08 | $5.8$ | $0M699$ | $1.9$ | $0M177$ |
| hole-09 | $80$ | $9M062$ | $16$ | $1M753$ |
| aim-100-1-6-sat-3 | $21.4$ | $3M173$ | $5.01$ | $0M505$ |
| aim-100-1-6-sat-4 | $160$ | $22M400$ | $47.1$ | $5M292$ |
| Renault-mod | | | | |
| renault-mod-15 | 505 | $1M969$ | 118 | $0M511$ |
| renault-mod-18 | $1,193$ | $6M812$ | 253 | $1M419$ |
| renault-mod-43 | 796 | $4M221$ | 23 | $0M127$ |
| renault-mod-45 | 85.5 | $0M591$ | 15.5 | $89,494$ |

## 8 Conclusion

In this paper, we have shown how values detected as globally inconsistent during search, and called failed values, can be useful to prune the search space through the introduction of a new class of consistencies that are orthogonal to the usual ones. Whereas FVC is a consistency that is cheap to check and that makes MAC more robust, AFVC and its direct extensions require further developments to determine the best way of controlling the enforcement of these new domain-filtering consistencies. We have also noticed that AFVC allows us to detect in a lazy manner a generalized form of neighborhood substitutability.

Although it is related to approaches that eliminate redundancies by posting constraints (for SAT) [21, 14] or decomposing problems [13], this way of reasoning has been developed so as to yield a hierarchy of increasingly stronger consistencies. For

[4] See `http://www.cril.univ-artois.fr/CPAI08/`

binary constraint networks, failed values basically permit us to identify and exploit a form of nogood in the same spirit as generalized nogood in [15], global cut seed in [9], kernel [22] and partial state [18]. We believe that identifying common properties to these different approaches is an exciting perspective.

## References

1. K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. H. Bennaceur, C. Lecoutre, and O. Roussel. A decomposition technique for solving Max-CSP. In *Proceedings of ECAI'08*, pages 500–504, 2008.
3. C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.
4. C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 72(6-7):800–822, 2008.
5. M.C. Cooper. Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90:1–24, 1997.
6. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
7. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
8. R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.
9. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.
10. E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.
11. E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.
12. E.C. Freuder and C. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96*, pages 202–208, 1996.
13. E.C. Freuder and P.D. Hubbe. Using inferred disjunctive constraints to decompose constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 254–261, 1993.
14. G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1):45–61, 1989.
15. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.
16. C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.
17. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.
18. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Exploiting past and future: Pruning by inconsistent partial state dominance. In *Proceedings of CP'07*, pages 453–467, 2007.
19. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
20. U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
21. P.W. Purdom. Solving satisfiability with less searching. *IEEE transactions on pattern analysis and machine intelligence*, 6(4):510–513, 1984.
22. I. Razgon and A. Meisels. A CSP search algorithm with responsibility sets and kernels. *Constraints*, 12(2):151–177, 2007.
23. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.