Tutoriel : Créer sa propre contrainte globale

by Guillaume Rochart

Table of contents

1 Sémantique de la contrainte.... 2
2 Implémentation d'une contrainte sans explication....2
2.1 Les données....2
2.2 Les événements sur la contrainte....3
2.2.1 La méthode awake....3
2.2.2 La méthode propagate....4
2.3 Les événements sur les variables....5
2.3.1 Les méthodes awakeOnInf et awakeOnSup....5
2.3.2 La méthode awakeOnRem....6
2.3.3 La méthode awakeOnInst....6
2.4 La construction de la contrainte....7
2.5 Quelques méthodes supplémentaires....8
3 Implémentation d'une contrainte avec explications....8
3.1 Expliquer le filtrage....8
3.2 Maintien des structures....9
3.3 Exécution du filtrage....11
3.4 Les réveils lors de la restauration....12
3.5 Quelques autres méthodes....12

Le but de ce tutoriel est de présenter le fonctionnement interne d'une contrainte globale et son implémentation. Comme illustration, nous prendrons la contrainte `occur` qui permet de contraindre le nombre d'apparitions d'une valeur dans un ensemble de variales.

1. Sémantique de la contrainte

Nous commençons dans cette partie par préciser la sémantique souhaitée pour cette contrainte. Cette contrainte a pour paramètres :

- un ensemle de variables (x1...xn) dont une variable particulière xn notée aussi nbOcc;
- une valeur étudiée v : il s'agit de la valeur dont on contraint le nombre d'occurences;
- deux booléens spécifiant si l'on contraint le nombre d'ocurrences minimal, maximal, ou encore les deux.

Si l'on souhaite contraindre le nombre minimal d'occurences de la valeur v parmis x1...xn-1, il faut vérifier à chaque moment qu'il y a toujours suffisamment d'occurences de v dans les valeurs *possibles* des domaines associés aux variables. De même, si l'on souhaite contraindre le nombre maximal d'occurences, il faut vérifier le nombre d'occurences *nécessaires*, c'est-à-dire le nombre de variables déjà instanciées à la valeur v. Dans le cas où l'on souhaite contraindre les deux bornes, il faut utiilser ces deux raisonnments.

Par exemple :

- si Dx1={1,2,4}, Dx2={2,3}, Dx3={1,4} et DnbOcc={1,3}, et que l'on cherche à contraindre le nombre d'occurences minimal de la valeur v=3, il est claire qu'il ne reste qu'une seule variable pouvant être affectée à v, on peut donc en inférer x2=3.
- si maintenant Dx1={3}, Dx2={1,3,4}, Dx3={3} et DnbOcc={1,2), et que l'on cherche à contraindre le nombre d'occurences maximal de la valeur v=3, on peut inférer que x2#3.

2. Implémentation d'une contrainte sans explication

Pour simplifier, nous commençons par décrire l'implémentation de la contrainte `occur` dans le cas non expliqué. Comme nous le verrons, ajouter des explications dans des contraintes globales peut demander du travail supplémentaire. S'agissant d'une contrainte prenant en compte de nombreuses variables, on peut créer la classe en étendant `AbstractLargeIntConstraint` qui permet d'implémenter de telles contraintes.

```
public class Occurence extends AbstractLargeIntConstraint {
```

2.1. Les données

Pour obtenir des contraintes efficaces, il est nécessaire de rendre les algorithmes incrémentaux (il est impensable de tout recalculer suite à chaque décision prise par l'algorithme de recherche). Nous devons ici stocker les informations suivantes : quels sont les variables qui peuvent ou doivent être affectées à la valeur v qui nous intéresse, et quel est le nombre de variables pouvant ou devant prendre cette valeur.

De plus, pour assurer la consistance des données lors d'un retour arrière de l'algorithme de recherche, nous utiliserons les types *storés* que nous offre Choco :

```
  public StoredBitSet isPossible; // v appartient-elle à x ?
  public StoredBitSet isSure; // x est-elle instanciée à v
  public StoredInt nbPossible;
  public StoredInt nbSure;
```

Enfin, nous stockons dans des variables booléennes sur quelle(s) borne(s) il faut travailler :

```
  public boolean constrainOnInfNumber;
  public boolean constrainOnSupNumber;
```

Le reste des données (et notamment les variables du problème) est directement géré par la super-classe `AbstractLargeIntConstraint`, il suffit donc dans le constructeur de mettre à jour ces variables.

2.2. Les événements sur la contrainte

La contrainte peut recevoir deux types d'évènements : les évènements liés à la contrainte et les événements liés à une variable en particulier. Nous présentons ici le premier type d'événements.

Les événements liés à la contrainte sont soit un premier réveil `awake` (ce qui donne l'occasion à contrainte d'initialiser certaines données et de faire une première propagation) soit un réveil classique `propagate`.

2.2.1. La méthode awake

Lors du premier réveil, il est possible sans calcul d'affirmer que le nombre d'occurences de la valeur v est compris entre 0 et le nombre de variables (mis à part la variable nbOcc). D'où la définition suivante :

```
  public void awake() throws ContradictionException {
    if (constrainOnInfNumber)
      vars[vars.length - 1].updateSup(vars.length -
1,cIndices[vars.length - 1]);
    if (constrainOnSupNumber)
      vars[vars.length - 1].updateInf(0,cIndices[vars.length - 1]);
    propagate();
  }
```

On notera que cette méthode peut lever une exception `ContradictionException` puisqu'elle modifie des domaines et peut donc entrainer un domaine vide. En effet, `updateInf` (resp. `updateSup`) permet de mettre à jour la valeur minimale (resp. maximale) d'une variable.

Comme le point fixe est atteint (si la contrainte est réveillée pour l'informer de ces modifications, elle ne déduira rien de plus), on précise à ces méthodes l'index de cette contrainte pour éviter qu'elle ne soit appelée suite à ces modifications. Les contraintes stockent ces indices dans le tableau `cIndices` ou sont accessibles via la méthode `getConstraintIdx(int)`. Dans le doute, on peut indiquer la valeur `-1` permettant à

la contrainte d'être réveillée suite à une modification dont elle est responsable : dans ce cas, on ne risque que de faire des calculs inutiles.

2.2.2. La méthode propagate

Il reste maintenant à implémenter la méthode `propagate` qui sera lancée lors du réveil de cette contrainte (un tel réveil peut être demandé grâce à la méthode `constAwake(boolean)` des contraintes). Ce réveil a souvent lieu pour mettre à jour les informations de la contrainte (lorsque celle-ci revient dans un état actif par exemple). Nous allons donc dans un premier mettre à jour les données, puis filtrer autant que possible les valeurs impossibles :

```
public void propagate() throws ContradictionException {
  for (int j = 0; j < vars.length - 1; j ++) {
    if (isPossible.get(j)) {
      if (!isSure.get(j) && vars[j].isInstantiatedTo(cste)) {
        isSure.set(j);
        nbSure.add(1);
      } else if (!vars[j].canBeInstantiatedTo(cste)) {
        isPossible.clear(j);
        nbPossible.add(-1);
      }
    }
  }
  filter();
}
```

Cette méthode permet de parcourir l'ensemble des variables (mis à part `nbOcc`) et de mettre à jour les données associées à ces variables (selon qu'elle soit instanciée à v ou qu'elle ne puisse pas être instanciée à v). Puis elle appelle une méthode `filter` qui ne fait pas partie de l'API d'une contrainte mais qui nous permet de traiter à part l'aspect filtrage (ce qui semble être une bonne idée lorsque l'on souhaite par la suite implémenter une version expliquée).

Pour cela, on ajoute les méthodes suivantes :

```
public void checkNbPossible() throws ContradictionException {
   int nbVars = vars.length - 1;
   if (constrainOnInfNumber) {
     vars[nbVars].updateSup(nbPossible.get(),cIndices[nbVars]);
     if (vars[nbVars].getInf() == nbPossible.get()) {
       for (int i = 0 ; i < nbVars ; i ++ ) {
        if (isPossible.get(i))  vars[i].instantiate(cste,cIndices[i]);
       }
     }
   }
}

public void checkNbSure() throws ContradictionException {
   int nbVars =  vars.length - 1;
   if (constrainOnSupNumber) {
     vars[nbVars].updateInf(nbSure.get(),cIndices[nbVars]);
     if (vars[nbVars].getSup() == nbSure.get()) {
       for (int i = 0 ; i < nbVars ; i ++ ) {
        if (isPossible.get(i) && !vars[i].isInstantiated())
```

```
                vars[i].removeVal(cste,cIndices[i]);
            }
          }
        }
    }

    public void filter() throws ContradictionException {
      checkNbPossible();
      checkNbSure();
    }
```

Les méthodes `checkNbPossible` et `checkNbSure` filtre les valeurs en utilisant les principes introduits dans la section sur la sémantique de la contrainte : si le nombre de variables *pouvant* être égal à v est égal au nombre minimal d'occurences que la contrainte doit accepter, alors on peut instancier ces variables.

2.3. Les événements sur les variables

Le deuxième style d'évenements que peut recevoir une contraint sont les événements liés aux variables (suite à des décisions prises par l'algorithme de recherche ou suite à un filtrage d'une autre contrainte). Les différentes méthodes pour gérer ces événements sont :

- `awakeOnInf` qui réagit sur la modification de la valeur minimale de la variable (elle doit toujours être définie selon la sémantique de la contrainte),
- `awakeOnSup` son symétrique pour la valeur maximale,
- `awakeOnRem` qui réagit sur un retrait d'une valeur entre les deux bornes (à redéfinir),
- `awakeOnInst` qui réagit sur une instantiation d'une de ses variables (à redéfinir),
- `awakeOnRemovals` qui réagit à plusieurs retraits de valeurs : par défaut elle parcourt tous les retraits de valeur de la variable en question et appelle `awakeOnRem`; elle ne nécessite donc pas d'être redéfinie *a priori*,
- enfin `awakeOnVar` qui réagit à une modification de variable sans précision (elle appelle par défaut `propagate`).

Warning:

La sémantique des ces méthodes n'est pas la même si des explications sont utilisées. Dans ce dernier cas, `awakeOnInst` et `awakeOnVar` ne sont pas utilisées, `awakeOnInf`/`awakeOnSup` ne sont utilisées qu'avec des variables de types `BoundIntVar` et `awakeOnRem`/`awakeOnRemovals` ne sont utilisées qu'avec des variables de type `EnumIntVar`. Ceci est dû notamment au stockage des explications pour ces variables.

Nous allons ici redéfinir les quatre première méthodes.

2.3.1. Les méthodes awakeOnInf et awakeOnSup

Ces deux méthodes sont très similaires : si les bornes modifiées ne permettent plus l'instantiation de la variable à la valeur v, il faut mettre à jour les données. De plus, si ces bornes deviennent égales à cette valeur v, il faut vérifier qu'il est possible d'affecter une variable de plus à v (notamment pour les variables de type `BoundIntVar` qui ne permettent pas de retirer les valeurs au milieu de leurs bornes), et dans le cas contraire la borne est encore décalée pour retirer cette valeur du domaine. Par exemple, dans le cas de

la borne inférieure, cela donne :

```
public void awakeOnInf(int idx) throws ContradictionException {
  int nbVars = vars.length - 1;
  if (idx < nbVars) {
    if (isPossible.get(idx)) {
      if (vars[idx].getInf() > cste) {
        isPossible.clear(idx);
        nbPossible.add(-1);
        checkNbPossible();
      } else if (vars[idx].getInf() == cste &&
             !(isSure.get(idx)) &&
             constrainOnSupNumber &&
             nbSure.get() == vars[nbVars].getSup()) {
        vars[idx].updateInf(cste + 1, cIndices[idx]);
      }
    }
  } else
    checkNbPossible();
}
```

De plus, dans le cas où c'est nbOcc qui est modifiée, on appelle `checkNbPossible` pour vérifier qu'il y a toujours assez de variables pouvant être affectées à v (dans le cas de la borne supérieure, la méthode `checkNbSure` aurait été appelée).

2.3.2. La méthode awakeOnRem

Cette méthode est plus simple : s'il s'agit d'un retrait de la valeur v, il faut mettre à jour les données, et vérifier qu'il y a toujours suffisamment de variables pouvant être instanciées à v.

```
public void awakeOnRem(int idx, int x) throws ContradictionException {
  int nbVars = vars.length - 1;
  if (idx < nbVars && x == cste && isPossible.get(idx)
    && vars[idx].hasEnumeratedDomain()) {
    isPossible.clear(idx);
    nbPossible.add(-1);
    checkNbPossible();
  }
}
```

La condition `vars[idx].hasEnumeratedDomain()` permet tout simplement de vérifier qu'il ne s'agit pas d'une variable de type `BoundIntVar` car dans ce cas, il ne faut pas prendre en compte cet événement puisqu'elle ne peut pas stocker les informations à propos des retraits de valeurs entre les deux bornes. De plus, s'il s'agit de la variable nbOcc, il n'y a rien à faire car nous ne nous intéressons uniquement aux bornes de cette variable.

2.3.3. La méthode awakeOnInst

Cette dernière méthode est un mélange de `awakeOnInf` et `awakeOnSup` : si la valeur v était possible mais pas sur, il faut vérifier si la variable est instanciée à v ou non pour mettre à jour correctement les données. S'il s'agit de la variable nbOcc (et non de xi), il faut vérifier qu'il y a suffisamment de variables pouvant être affectée à v et pas trop déjà

instantiées à v :

```
  public void awakeOnInst(int idx) throws ContradictionException {
    int nbVars = vars.length - 1;
    if (idx < nbVars && isPossible.get(idx) && !(isSure.get(idx))) {
      if (vars[idx].getValue() == cste) {
        isSure.set(idx);
        nbSure.add(1);
        checkNbSure();
      } else {
        isPossible.clear(idx);
        nbPossible.add(-1);
        checkNbPossible();
      }
    } else {
      checkNbPossible();
      checkNbSure();
    }
  }
```

2.4. La construction de la contrainte

Pour créer la contrainte, il faut bien entendu fournir un constructeur permettant d'initialiser variables, données, etc :

```
  public Occurence(IntVar[] lvars, int occval, boolean onInf, boolean
onSup) {
    super(lvars.length);
    this.cste = occval;  // L valeur étudiée v
    this.constrainOnInfNumber = onInf;  // Propagation sur le nombre
minimal
    this.constrainOnSupNumber = onSup;  // Propagation sur le nombre
maximal

    this.problem = lvars[0].getProblem();  // Stocke le problème associé

    int nbVars = lvars.length;
    // Détermine l'environnement utilisés pour  les données storées
    Environment envi = lvars[0].getProblem().getEnvironment();
    isPossible = new StoredBitSet(envi, nbVars - 1);
    isSure = new StoredBitSet(envi, nbVars - 1);
    nbPossible = new StoredInt(envi, 0);
    nbSure = new StoredInt(envi, 0);

    // Affectation des variables et mis à jour des données
    for (int i = 0; i < (nbVars - 1); i++) {
      vars[i] = (IntDomainVar) lvars[i];
      if (vars[i].canBeInstantiatedTo(occval)) {
        isPossible.set(i);
        nbPossible.add(1);
      }
      if (vars[i].isInstantiatedTo(occval)) {
        isSure.set(i);
        nbSure.add(+1);
      }
    }
    vars[nbVars - 1] = (IntDomainVar) lvars[nbVars - 1];
  }
```

2.5. Quelques méthodes supplémentaires

Quelques méthodes peuvent être encore nécessaire, comme `isSatisfied` qui permet de vérifier si la contrainte est vérifiée (une fois toutes les variables instantiées). Il suffit ici de compter le nombre d'apparitions de la valeur v sur l'ensemble des variables :

```
public boolean isSatisfied() {
  int nbVars = vars.length - 1;
  int cptVal = 0;
  for (int i = 0; i < nbVars; i++) {
    if(vars[i].getValue() == cste) cptVal++;
  }
  if (constrainOnInfNumber && constrainOnSupNumber)
    return cptVal == vars[nbVars].getValue();
  else if (constrainOnInfNumber)
    return cptVal >= vars[nbVars].getValue();
  else return cptVal <= vars[nbVars].getValue();
}
```

3. Implémentation d'une contrainte avec explications

L'implémentation d'une contrainte avec explications demandent plus d'information. Nous ne détaillerons donc pas l'implémentation complète de la contrainte. Dans la version classique (sans explications), le filtrage a bien été séparé du reste dans les méthodes `checkNbPossible` et `checkNbSure`; nous devrons donc principalement modifier ces méthodes de sorte à y générer les explications justifiant les décisions prises. De plus, les contraintes expliquées sont utilisées dans un cadre dynamique : des contraintes sont ajoutées et d'autres retirées, ce qui nécessite de maintenir les données. Il existe donc maintenant deux types de méthodes réagissant sur les événements liés aux variables : les méthodes modifiant les données `updateDataStructuresOnX` et les méthodes lançant le filtrage en lui-même. Nous verrons comment implémenter certaines d'entre elles. Enfin, des méthodes utiles peuvent (et devraient) être implémentées comme `whyIsTrue` ou `whyIsFalse`.

Warning:

Nous supposons ici que l'on souhaite hériter de la contrainte sans explication. Ceci n'est pas obligatoire, mais peut être utile lorsque l'on ne souhaite pas dupliquer la structure de donnée (qui peut être complexe dans certains cas !).

3.1. Expliquer le filtrage

Pour utiliser les explications, il faut pouvoir déterminer pour chaque valeurs filtrées ce qui permet de justifier ce retrait. Ainsi, si l'on sait quels propriétés (borne inférieure, supérieure, valeur absente d'un domaine...) sont à l'origine du retrait, il est possible de générer une explication en faisant l'union des explications de ces propriétés (toute modification d'un domaine d'une variable stocke une explication permettant de justifier celle-ci).

Prenons le cas de `checkNbPossible`. La première chose que fait cette méthode, c'est

de mettre à jour la borne supérieure de nbOcc pour que celle-ci soit inférieure ou égale au nombre de variables pouvant être instanciées à v. Pour justifier cette valeur, il suffit de justifier le nombre de variables ne pouvant pas prendre cette valeur (le nombre de variables est fixe), c'est-à-dire de justifier que ces variables ne peuvent plus être instantiée à cette valeur v. De plus, l'explication doit contenir la contrainte en cours d'exécution qui est responsable de ce filtrage (grâce à la ligne suivant l'initialisation de l'explication dans le code). La méthode commencera donc comme suit :

```
  public void checkNbPossible() {
    if (this.constrainOnInfNumber) {
      Explanation expl = new GenericExplanation(problem);
      ((PalmConstraintPlugin) this.hook).self_explain(expl);

      for (int i = 0; i < this.vars.length - 1; i++) {
        if (!this.isPossible.get(i)) {
          ((PalmIntVar) this.vars[i]).
              self_explain(PalmIntDomain.VAL, this.cste, expl);
        }
      }

      ((PalmIntVar) this.vars[this.vars.length - 1]).updateSup(
          this.nbPossible.get(), cIndices[this.vars.length - 1],
expl.copy());
```

Ensuite, si ce nombre de variables pouvant être égal à v est égal au nombre minimal d'occurences que doit accepter la contrainte, alors il faut instantier toutes ces variables. L'explication sera alors la même si ce n'est qu'on lui ajoutera la justification de cette borne inférieure (celle de la variable nbOcc) :

```
      if (this.vars[this.vars.length - 1].getInf() ==
this.nbPossible.get()) {
        ((PalmIntVar) this.vars[this.vars.length - 1]).
            self_explain(PalmIntDomain.INF, expl);
        for (int i = 0; i < this.vars.length - 1; i++) {
          if (this.isPossible.get(i)) {
            ((PalmIntVar) this.vars[i]).instantiate(this.cste,
                cIndices[i], expl.copy());
          }
        }
      }

    }
    this.checkPossible = false;
  }
```

Nous expliquerons plus loin à quoi correspond la dernière ligne de cette méthode.

3.2. Maintien des structures

Dans le cas de retour arrière simple, il n'est pas gênant que les événements sur les variables ne soient pas traitées de manière synchrone : si une contradiction est levée, tous les événements sont nettoyés. Dans un cadre dynamique, ceci n'est plus possible : il est donc important de garantir que les structures de données sont toujours à jour. Pour cela, des méthodes sont appelés de manière synchrone dès qu'un domaine est modifié pour donner une chance aux contraintes de maintenir leurs structures de données.

Dans le cas présent, maintenir la structure de données revient à mettre à jour les variables `nbPossible`, `isPossible`... Si l'on considère le cas d'une modification de borne inférieure par exemple, si la borne devient supérieure à la valeur v il faut mettre à jour le nombre de variables possibles. De même si la variable est instanciée à la valeur v il faut mettre à jour le nombre de variables sures.

```
public void updateDataStructuresOnConstraint(int idx, int select,
    int newValue, int oldValue) {
  switch (select) {
    case PalmIntDomain.INF:
      if (idx < this.getNbVars() - 1) {
        if ((newValue > this.cste) && (this.isPossible.get(idx))) {
          this.isPossible.clear(idx);
          this.nbPossible.add(-1);
          this.checkPossible = true;
          if (this.isSure.get(idx)) {
            this.isSure.clear(idx);
            this.nbSure.add(-1);
            this.checkSure = true;
          }
        }

        if (this.vars[idx].isInstantiated()) {
          int val = this.vars[idx].getValue();
          if ((val == this.cste) && (!this.isSure.get(idx))) {
            this.isSure.set(idx);
            this.nbSure.add(1);
            this.checkSure = true;
          }
        }
      } else {
        this.checkSure = true;
        this.checkPossible = true;
      }
      break;
```

On notera deux points justifiant ce code :

- si la valeur n'est possible alors qu'elle était sûre, unne cotradiction sera levée; cependant il est important de maintenir la structure à jour, car rien ne garantit que l'on remettra cette valeur (il ne s'agit plus d'un retour arrire classique !),
- si la variable est instanciée à une autre valeur que v, alors soit cette valeur est plus grande que v et donc le nombre des variables possibles a été mis à jour, soit cette valeur est plus petite que v, et cela signifie qu'un événement `awakeOnSup` sera lancé aussi permettant de mettre à jour cette variable (il ne faut donc pas mettre à jour cette variable plusieurs fois !).

Les deux dernières lignes, entre autres, mettent en avant deux nouvelles variables. Ces variables permettent de gérer aussi rapidement que possible le filtrage en évitant de refaire des tests déjà effectués (si l'on sait que rien ne sera filtré, il est inutile d'essayer).

La deuxième étape consiste à en faire de même pour la restauationde valeurs. Dans ce cas, il faut incrémenter le nombre de variables possibles ou sures si nécessaire. De plus, si une variable n'est plus instantiée (on lui ajoute une deuxième valeur) il faut modifier le

nombre de variables sures. Ceci donne donc :

```
  public void updateDataStructuresOnRestoreConstraint(int idx, int
select,
      int newValue, int oldValue) {
    switch (select) {
      case PalmIntDomain.INF:
        if (idx < this.getNbVars() - 1) {
          if ((newValue <= this.cste) && (oldValue > this.cste)
              && (this.vars[idx].getSup() >= newValue)) {
            this.isPossible.set(idx);
            this.nbPossible.add(1);
            this.checkPossible = true;
            if (this.vars[idx].isInstantiated()) {
              int val = this.vars[idx].getValue();
              if ((val == this.cste) && (!this.isSure.get(idx))) {
                this.isSure.set(idx);
                this.nbSure.add(1);
                this.checkSure = true;
              }
            }
          }
          if ((this.isSure.get(idx))
              && (this.vars[idx].getInf() != this.cste)) {
            this.isSure.clear(idx);
            this.nbSure.add(-1);
            this.checkSure = true;
          }
        } else {
          this.checkPossible = true;
          this.checkSure = true;
        }
        break;
```

3.3. Exécution du filtrage

Le maintien des structures ayant spécifié les filtrages à lancer, le travail est mainteant beaucoup plus simple lors des réveils liés ax variables. Cependant, il reste des cas à traiter. Si l'on reprend le cas de la borne inférieure, si la borne devient égale à v, il est nécessaire de vérifier qu'il est possible que cette variable prenne cette valeur (c'est-à-dire qu'il n'y a pas déjà trop de variables instanciées à cette valeur) : il s'agit d'un filtrage de provenant pas directement de la structure de donnée, c'est pour cela qu'il est nécessaire de le traiter à part. On obtient alors le code suivant :

```
  public void awakeOnInf(int idx) {
    if (idx < this.getNbVars()) {
      if (this.checkPossible)
        this.checkNbPossible();
      if (this.checkSure)
        this.checkNbSure();
      if ((this.isPossible.get(idx)) &&
          (this.vars[idx].getInf() == this.cste))
        if ((!this.isSure.get(idx)) && (this.constrainOnSupNumber) &&
            (this.nbSure.get() == this.vars[this.vars.length -
1].getSup())) {
          Explanation expl = new GenericExplanation(problem);
          ((PalmConstraintPlugin) this.hook).self_explain(expl);
```

```
          ((PalmIntVar) this.vars[idx]).
              self_explain(PalmIntDomain.INF, expl);
          ((PalmIntVar) this.vars[this.vars.length - 1]).
              self_explain(PalmIntDomain.SUP, expl);
          for (int i = 0; i < this.getNbVars() - 1; i++) {
            if (this.isSure.get(i))
              ((PalmIntVar) this.vars[i]).
                  self_explain(PalmIntDomain.DOM, expl);
          }
          ((PalmIntVar) this.vars[idx]).
              updateInf(this.cste + 1, cIndices[idx], expl);
        }
    } else {
      this.checkNbPossible();
    }
  }
```

Le même style de code devra être développé pour `awakeOnSup`, `awakeOnRem` ou `awakeOnRemovals`.

3.4. Les réveils lors de la restauration

Lorsqu'une contrainte est retirée, certaines valeurs peuvent être restaurées dans les domaines des variables. Il faut donc vérifier si ces valeurs sont consistantes avec la contrainte. Ici, il est difficile de savoir quelle partie du filtrage doit être vérifiée, les fonctions appelle donc la méthode `filter()` qui elle-même vérifie à la fois les propriétés sur le nombres de variables possibles et le nombre de nécessaires :

```
  public void awakeOnRestoreInf(int idx) throws ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreSup(int idx) throws ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreVal(int idx, int val) throws
ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreVal(int idx, IntIterator it)
      throws ContradictionException {
    for (; it.hasNext();) {
      awakeOnRestoreVal(idx, it.next());
    }
  }
```

On notera que puisque l'on hérite de la version non expliquée, il est nécessaire de définir une méthode `awakeOnRestoreVal(int, IntIterator)` puisqu'aucune superclasse ne la définit.

3.5. Quelques autres méthodes

Outre la déclaration des données supplémentaires, la construction de la contrainte version

expliquée ne demande pas d'information particulière si ce n'est l'initialisation d'un *plugin* permettant de stocker les informations relatifs aux explications :

```
  private boolean checkPossible = false, checkSure = false;

  public PalmOccurence(IntVar[] lvars, int occval, boolean onInf,
boolean onSup) {
    super(lvars, occval, onInf, onSup);
    this.hook = new PalmConstraintPlugin(this);
  }
```

Outre les méthodes ues jusqu'ici (obligatoires pour le bon fonctionnement de la contrainte), on peut définir les fonctions `whyIsTrue` et `whyIsFalse` permettant de savoir pourquoi une contrainte est forcément fausse, ou pourquoi elle est forcément vérifiée.

Enfin, comme il s'agit d'une contrainte expliquée, la classe implémente les interfaces suivantes : `PalmConstraint` et `PalmIntVarListener`.

FIXME (Guillaume):

Voir si on parle de `takeIntoAccountStatusChange(int)` !?