MCSP3: la modélisation pour tous

Christophe Lecoutre

CRIL-CNRS UMR 8188,
Université d'Artois, F-62307 Lens, France
lecoutre@cril.fr

Résumé

Nous proposons MCSP3, une API de modélisation pour les problèmes combinatoires sous contraintes, actuellement "limitée" aux problèmes de satisfaction de contraintes (CSP) et problèmes d'optimisation sous contraintes (COP). MCSP3 est accompagné d'un compilateur qui permet de générer des instances définies au format intermédiaire XCSP3. La prise en main de cette API est rendue simple et naturelle, car elle est construite sur Java, sans doute aujourd'hui le langage le plus populaire en informatique. Grâce aux nouveaux concepts de la version 8 de Java, et notamment les lambda fonctions, l'écriture des modèles est compacte, particulièrement lisible, et pour l'aspect formel relativement proche des meilleurs langages dédiés. L'API propose de nombreuses méthodes, évitant dans une large mesure toute expression idiomatique (technique) propre au langage telle que l'utilisation de `new` pour la création d'objets, de tableaux, etc. Sur la base d'une expérience correspondant au développement de près d'une centaine de modèles à ce jour, nous pensons sincèrement que cette API est accessible au plus grand nombre. MCSP3 est une API de modélisation pour tous, disponible sur github.

1 Introduction

Le monde de la programmation par contraintes (PPC) pâtit du manque de standards, que ce soit pour la représentation des modèles ou pour celle des instances de problèmes combinatoires sous contraintes. Un modèle est une représentation paramétrée de la structure d'un ensemble d'instances pour un problème donné. Par exemple, il est possible de développer un modèle pour le problème des n-reines, où n représente une valeur entière jouant le rôle de paramètre, et de s'intéresser à la résolution d'une instance précise de ce problème, comme par exemple l'instance des 8-reines. Depuis trois ans, au sein du CRIL [1], nous développons une chaîne d'outils qui se revendique in fine être une approche globale, naturelle et cohérente permettant la modélisation et la résolution de problèmes sous contraintes. Dans cet article, nous présentons une brique importante de cette chaîne : MCSP3, une API de modélisation à la portée de tous.

Intéressons nous tout d'abord aux langages et outils de modélisation introduits dans la littérature au fil du temps. Parmi les langages marquants, certains sont dédiés à la programmation mathématique, comme AMPL (http://www.ampl.com) et GAMS (http://www.gams.com), et d'autres [2] à la programmation par contraintes, comme OPL [15], EaCL [14], NCL [16], ESRA [6], Zinc [4] et ESSENCE [7]. À ce jour, la situation reste malheureusement assez confuse pour l'utilisateur qui souhaite adopter une solution de modélisation PPC satisfaisante et pérenne.

Tout avait pourtant bien commencé. En effet, au début des années 70 [3], un langage étonnant voir le jour : Prolog. L'originalité de ce langage, basé sur le calcul des prédicats du premier ordre, son élégance et son aspect déclaratif engendre une grande effervescence, et ceci pendant de nombreuses années. La version 3 du langage [2] adopte un cadre de programmation logique par contraintes, et obtient un large succès, y compris auprès du grand public [3]. Au fil du temps, l'aspect "contraintes" se substitue à l'aspect "logique", et de nouveaux produits sont développés, les plus emblématiques étant certainement CHIP [5] et ILOG Solver. En terme de modélisation, OPL (Optimization Programming Language) [15] apparaît sans doute à ce

1. Avec le concours appréciable de collègues rattachés à des laboratoires de France et de Navarre (et aussi de Belgique ;-)).
2. Le langage Z (http://vl.users.org) permet également le développement de modèles plutôt élégants [11].
3. Il était fréquent que Prolog soit enseigné dans les formations universitaires en informatique.

moment-là comme étant le langage plus abouti, mais son caractère propriétaire le prive d'une appropriation par l'ensemble de la communauté.

Après les belles aventures Prolog et OPL, la communauté se tourne progressivement vers le développement de solveurs ouverts, le plus souvent intégrant une interface de modélisation dédiée, tels que par exemple Gecode [13] et Choco [10]. Cependant, le besoin d'uniformisation se faisant sentir de plus en plus fortement, de nouveaux langages de modélisation sont proposés, notamment au cours de la dernière décennie avec Essence et MiniZinc. Si Essence est assurément de belle facture, la suite qui l'accompagne reste, nous semble-t-il, assez difficile d'accès. En ce qui concerne Minizinc, nous n'avons jamais été convaincus par les choix effectués par les développeurs de cette suite (nous développerons nos arguments dans un article à venir).

Pour la représentation d'instances de problèmes, il est possible de se tourner vers des formats à plat tels que XCSP 2.1 [12] ou FlatZinc, ou le format intermédiaire XCSP3 [1] que nous avons introduit récemment. XCSP3 a l'avantage d'être très complet, lisible et structuré. La figure 1 offre une vision synthétique de la situation actuelle concernant langages et formats de modélisation.

FIGURE 1 – Langages de modélisation et formats

Ce constat posé, nous avons cherché à développer de façon posée et cohérente une chaîne de production complète et intégrée pour le domaine de la PPC. Les deux principaux ingrédients sont :

— MCSP3 : une interface (API) basée sur Java permettant de modéliser les problèmes sous contraintes de façon déclarative et naturelle ;
— XCSP3 : un format intermédiaire utilisé pour représenter les instances de problèmes, tout en préservant leurs structures.

Comme cela est visible sur la figure 2, confronté à un problème à résoudre, l'utilisateur doit procéder comme suit :

1. écrire un modèle en utilisant l'interface MCSP3 construite sur Java 8 ;
2. fournir les données, au format JSON, correspondant aux instances précises à résoudre ;
3. compiler modèle et données afin de générer des fichiers au format intégré XCSP3 ;
4. résoudre les instances XCSP3 en utilisant des solveurs tels que Choco ou OscaR [9].

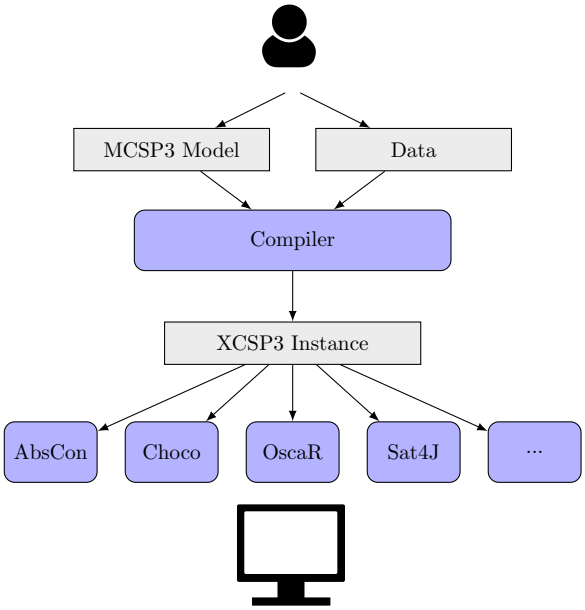

FIGURE 2 – Chaîne de production complète MCSP3-XCSP3.

La chaîne de production complète, MCSP3-XCSP3, a de nombreux avantages :

— Java, JSON et XML sont des technologies éprouvées de premier plan ;
— utiliser Java 8 pour la modélisation [4] évite à l'utilisateur d'apprendre encore (et toujours) un nouveau langage ;
— utiliser JSON pour les données offre une notation simple, unifiée et facile à lire aussi bien pour l'humain que pour la machine :
— utiliser XML pour les instances (mais avec une granularité raisonnable) permet une représentation simple et lisible, de nouveau que ce soit pour l'humain ou la machine.

Il peut sembler étonnant que JSON et XML soient tous les deux sollicités le long de la chaîne. En fait, nous sommes convaincus que JSON est le format le plus approprié pour les données tandis que XML est

4. L'introduction des lambda fonctions en Java 8 a été l'élément déclencheur qui nous a permis de développer une interface sobre et naturelle.

mieux adapté pour la représentation des instances (utiliser JSON est envisageable, mais possède quelques inconvénients comme cela est indiqué dans l'une des appendices des spécifications XCSP3 [1]).

La version courante de MCSP3 est une "release candidate", mais elle est néanmoins déjà robuste car testée sur une centaine de modèles. L'API (avec son compilateur) est disponible sur github : https://github.com/xcsp3team/XCSP3-Java-Tools, dans le package `modeler`. Une documentation plus fournie se trouve dans [8].

2 Illustration

Nous proposons d'illustrer l'interface MCSP3 avec un cas d'étude, le problème d'allocation d'entrepôt défini sur CSPLib–Problem 034 comme suit :

"In the Warehouse Location problem (WLP), a company considers opening warehouses at some candidate locations in order to supply its existing stores. Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply. Each store must be supplied by exactly one open warehouse. The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized."

FIGURE 3 – Un entrepôt.

Du point de vue de l'utilisateur, modéliser un problème nécessite trois étapes :

1. Définir la structure (type) des paramètres du problème. Une instance du problème est-elle caractérisée par un simple entier, une séquence d'entiers, un ou plusieurs tableaux de valeurs numériques, voire symboliques ? etc.
2. Définir la structure du modèle en utilisant un langage approprié. Ici, il s'agit de MCSP3.
3. Définir un jeu d'instances en produisant un jeu de données respectant la structure préalablement définie.

Après réflexion, nous découvrons que notre problème nécessite trois paramètres. Le premier `fixedCost` représente le coût fixe entier associé à l'ouverture d'un entrepôt. Le second, `warehouseCapacities` représente le nombre de magasins que chaque entrepôt peut alimenter. Le troisième, `storeSupplyCosts` représente le coût pour alimenter chaque magasin avec chaque entrepôt. La structure des paramètres est donc donnée ici par un entier, un tableau d'entiers et un tableau à deux dimensions d'entiers. Un exemple de données pour une instance spécifique est donné par le fichier suivant "warehouse.json" contenant :

```
{
  "fixedCost": 30,
  "warehouseCapacities": [1,4,2,1,3],
  "storeSupplyCosts": [
    [100,24,11,25,30],[28,27,82,83,74],
    [74,97,71,96,70],[2,55,73,69,61],
    [46,96,59,83,4],[42,22,29,67,59],
    [1,5,73,59,56],[10,73,13,43,96],
    [93,35,63,85,46],[47,65,55,71,95]
  ]
}
```

Un modèle possible en MCSP3 est :

```
class Warehouse implements ProblemAPI {
  int fixedCost;
  int[] warehouseCapacities;
  int[][] storeSupplyCosts;

  public void model() {
    int nWarehouses = warehouseCapacities.length;
    int nStores = storeSupplyCosts.length;

    Var[] s = array("s", size(nStores),
      dom(range(nWarehouses)),
      "s[i] is the warehouse supplier of store i");

    Var[] c = array("c", size(nStores),
      i -> dom(storeSupplyCosts[i]),
      "c[i] is the cost of supplying store i");

    Var[] o = array("o", size(nWarehouses),
      dom(0, 1), "o[i] is 1 if the warehouse i is open");

    forall(range(nWarehouses),
      i -> atMost(s, i, warehouseCapacities[i]));
    forall(range(nStores),
      i -> element(o, s[i], 1));
    forall(range(nStores),
      i -> element(storeSupplyCosts[i], s[i], c[i]));

    int[] coeffs = vals(repeat(1,nStores),
       repeat(fixedCost,nWarehouses));
    minimize(SUM, vars(c,o), coeffs)
      .note("minimizing the overall cost");
  }
}
```

Comme vous pouvez l'observer, définir un modèle nécessite donc d'écrire une classe implémentant l'interface `ProblemAPI`. Cette classe doit intégrer des champs correspondant aux données du fichier JSON (le chargement se fait automatiquement au moment de la compilation). Il est ensuite nécessaire d'implanter la méthode `model()`, en intégrant la déclaration des variables, contraintes et objectifs du problème. Pour notre problème, nous déclarons trois tableaux de variables : la taille des tableaux est définie par appel à la méthode `size()`, et le domaine de chaque variable est défini par appel à la méthode `dom()`. Il est intéressant de constater qu'il est possible de définir un domaine spécifique pour chaque variable d'un tableau en utilisant une lambda fonction. Pour poster des groupes de contraintes, on utilise la méthode `forall`. Le premier groupe signifie que pour tout entrepôt d'indice i, on souhaite qu'au plus `warehouseCapacities[i]` variables de s prennent la valeur i. Les deux autres groupes permettent de poster des contrainte `element`. Pour finir, l'objectif à minimiser est une somme de variables coefficientées. La méthode `vars()` permet de définir un tableau de variables en collectant celles qui sont passées en paramètres (y compris lorsqu'elles sont déjà définies dans des tableaux).

De façon générale, il existe dans l'API de nombreuses surcharges de méthodes pour définir les tableaux de variables (jusqu'à une dimension 5), les domaines, les groupes (jusque 5 boucles imbriquées en utilisant la combinaison de méthodes `range()`). Actuellement, dans la version courante (release candidate), la vingtaine de contraintes appartenant à XCSP3-core est utilisable. Cela offre déjà un grand pouvoir d'expression, et nous a d'ailleurs permis de modéliser une centaine de problèmes.

3 Conclusion

MCSP3 est une API de modélisation qui permet de représenter les problèmes combinatoires sous contraintes. Basée sur Java 8, elle permet à l'utilisateur de construire aisément les modèles et de les compiler en instances XCSP3. Les versions futures intégreront d'autres cadres (WCSP, par exemple), d'autres contraintes, les variables réelles et ensemblistes, la réification, les annotations, etc. Comparé à la "release candidate" courante de MCSP3, qui a nécessité un effort de développement particulièrement important, ces futures extensions ne présentent pas de difficultés techniques majeures.

Références

[1] F. Boussemart, C. Lecoutre, and C. Piette. XCSP3 : An integrated format for benchmarking combinatorial constrained problems. Technical Report arXiv :1611.03398, CRIL, 2016.

[2] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7) :69–90, 1990.

[3] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un système de communication homme-machine. Technical report, Groupe de recherche en Intelligence Artificielle, Marseille, 1973.

[4] M. Garcia de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language Zinc. In *Proceedings of CP'06*, pages 700–705, 2006.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of FGCS'88*, pages 693–702, 1988.

[6] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR'03 : Revised Selected Papers*, pages 214–232, 2004.

[7] A. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE : A constraint language for specifying combinatorial problems. In *Proceedings of IJCAI'07*, pages 80–87, 2007.

[8] C. Lecoutre. MCSP3 : Easy modeling for everybody. Technical Report Release Candidate, available from https://github.com/xcsp3team/XCSP3-Java-Tools, CRIL, 2017.

[9] OscaR Team. OscaR : Scala in OR, 2012. Available from https://bitbucket.org/oscarlib/oscar.

[10] C. Prud'homme, J.-G. Fages, and X. Lorca. Choco, 2016. Available from http://www.choco-solver.org.

[11] G. Renker and H. Ahriz. Building models through formal specification. In *Proceedings of CPAIOR'04*, pages 395–401, 2004.

[12] O. Roussel and C. Lecoutre. XML representation of constraint networks : Format XCSP 2.1. Technical Report arXiv :0902.2362, CoRR, 2009.

[13] C. Schulte, G. Tack, and M.Z. Lagerkvist. Gecode : A generic constraint development environment. http ://www.gecode.org, 2006.

[14] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In *Proceedings of PACLP'99*, pages 81–93, 1999.

[15] P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[16] J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1-3) :71–103, 2000.