Tests à données aléatoires par mutations pour les systèmes de programmation par contraintes

Wout Vanroose[1], Ignace Bleukx[1], Jo Devriendt[1], Dimos Tsouros[1], Hélène Verhaeghe[1,2], Tias Guns[1]

[1] KULeuven, Louvain, Belgique
[2] UCLouvain, Louvain-la-Neuve, Belgique

wout.vanroose@kuleuven.be,ignace.bleukx@kuleuven.be,jo.devriendt@kuleuven.be,
dimos.tsouros@kuleuven.be,helene.verhaeghe@uclouvain.be, tias.guns@kuleuven.be

Résumé

Les langages de modélisation de programmation par contraintes (PPC), tels que MiniZinc, Essence et CPMpy, jouent un rôle dans l'accessibilité des techniques de PPC aux non-experts. Les langages de modélisation, comme les solveurs, sont des pièces complexes de logiciels pouvant contenir des bogues, réduisant leur utilité. Les tests à données aléatoires par mutations sont une manière de tester des systèmes complexes en mutant des entrées et en vérifiant la préservation de propriétés dans la sortie mutée. Ce papier est un résumé de l'article "Mutational Fuzz Testing for Constraint Modeling Systems"[3] publié à CP2024.

Mots-clés

Tests à données aléatoires par mutations, langage de modélisation de problèmes PPC, bogue

Abstract

Constraint programming (CP) modeling languages, like MiniZinc, Essence and CPMpy, play a crucial role in making CP technology accessible to non-experts. Both solver-independent modeling frameworks and solvers themselves are complex pieces of software that can contain bugs, which undermines their usefulness. Mutational fuzz testing is a way to test complex systems by stochastically mutating input and verifying preserved properties of the mutated output. This paper is a summary of the article "Mutational Fuzz Testing for Constraint Modeling Systems"[3] published at CP2024.

Keywords

Fuzz testing, Constraint modeling language, bugs.

1 Introduction

La résolution sous contraintes est une technique de raisonnement d'IA déclarative qui est utilisée pour une variété d'applications allant de la planification de lignes de productions à l'automatisation de programmes informatiques en passant par des applications aérospatiales. Toutes ces applications requièrent que les solveurs de contraintes donnent des solutions correctes et fiables vis-à-vis du modèle.

Pour utiliser la puissance des solveurs modernes, il est commun pour les utilisateurs d'écrire les spécifications de leurs problèmes dans un langage de modélisation de contraintes haut niveau tel que MiniZinc, XCSP, Essence ou CPMpy[1]. Ces langages de modélisation jouent un rôle fondamental pour l'adoption à large échelle des technologies de programmation par contraintes (PPC) à travers de nombreux domaines grâce à leur formalisme haut niveau, leur expressivité et les méthodes de définition de problèmes intuitives pour les utilisateurs. Ils offrent une abstraction des détails de l'encodage des contraintes de haut niveau dans les contraintes spécifiques prises en charge par un solveur, permettant aux utilisateurs de se concentrer sur le problème en cours plutôt que sur les spécificités des solveurs.

Les systèmes de modélisation reformulent ensuite les contraintes utilisateur de haut niveau en des expressions spécifiques aux solveurs telles que des clauses, des contraintes linéaires ou des contraintes globales non imbriquées. Pour cette étape, le code de base des systèmes de modélisation inclut un certain nombre de reformulations et d'algorithmes d'encodage. Ces systèmes sont également complexifiés par la présence d'optimisations de modèles tels que les systèmes d'élimination de sous-expression commune.

Comme de nombreux logiciels complexes, les systèmes de modélisation et les solveurs de contraintes peuvent contenir des bogues. Dans le cas des systèmes de modélisation, ceux-ci peuvent avoir plusieurs types de comportements indésirables allant de l'interruption de fonctionnement du système à la génération de fausses solutions ou du retrait de solutions valides (ex : retrait de l'optimal). Ces situations peuvent avoir un impact important sur l'utilisateur et son application. De plus, cela peut diminuer la confiance des utilisateurs envers un solveur.

Pour ce faire, nous proposons le logiciel cadre *Hurricane*, basé sur les techniques de vérification automatique par données aléatoires par mutations[2] pour vérifier les systèmes de modélisation de programmation par contraintes.

2 Tests à données aléatoires par mutations

Notre système de tests à données aléatoires par mutations fonctionne de la manière suivante. Un ensemble de *m* problèmes de satisfaction ou d'optimisation de contraintes satisfaisables est sélectionné. A chaque itération de l'algorithme, un modèle est sélectionné de manière aléatoire et un certain nombre de mutations sont appliquées aux contraintes de ce modèle. Une mutation est une fonction qui prend en entrée un ensemble de contraintes et en sortie donne un autre ensemble de contraintes. Ces nouvelles contraintes sont ajoutées au modèle. Après application des mutations, nous vérifions si certaines propriétés (dépendantes du type de mutation) sont conservées. Si la vérification échoue, l'algorithme a découvert un bogue dans le système et l'enregistre pour analyse par l'utilisateur.

2.1 Ensemble de modèle de départ

Avoir des modèles simples permet une plus grande facilité à identifier la source en cas d'erreur. Pour également diminuer la complexité de création de cet ensemble et s'assurer que tous les types de contraintes supportées sont utilisées, la meilleure piste est de partir des modèles utilisés dans les tests unitaires. Ces modèles sont généralement petits, compréhensibles et contiennent un nombre restreint de contraintes et variables à la fois.

2.2 Mutations

Les mutations sont de plusieurs types, classées en fonction des propriétés qu'elles respectent mais également du type de modifications appliquées sur les contraintes.

2.2.1 Reformulations

Les systèmes de modélisation possèdent généralement plusieurs fonctions de reformulations afin de transformer les modèles définis par les utilisateurs vers les solveurs. Ces reformulations incluent entre autre la transformation en contraintes linéaires, la décomposition de contraintes globales ou la simplification de termes. Ces transformations peuvent servir directement de mutation.

2.2.2 Mutations haut niveau

Les mutations haut niveau sont des mutations basées sur la logique entre deux expressions booléennes. Par exemple, si le modèle contient deux contraintes A et B, la contrainte $A \wedge B$ est validée pour toutes les solutions du modèle initial, de même que pour $A \vee B$. Ajoutée au modèle, les mutations de ce type ne change pas l'ensemble de solutions acceptées.

2.2.3 Mutation de sous-expressions : fusion sémantique

L'idée de ce type de mutation est de fusionner deux expressions et créer une variable auxiliaire pour celle-ci. Par exemple, si on a les variables x_1 et x_2, on peut définir $y = x_1 + x_2$. Cette nouvelle expression est alors utilisée pour substituer les anciennes expressions là où elles sont utilisées. Dans le cas de l'exemple, x_1 sera substitué par $y - x_2$ et x_2 sera substitué par $y - x_1$.

2.2.4 Mutation de sous-expressions : comparaisons équivalentes

Ce type de mutation sélectionne une comparaison de manière aléatoire et applique une multiplication par une constante (positive) de chaque coté de la comparaison.

2.3 Méthodes de vérifications

Une fois le modèle muté obtenu, il faut vérifier si un bogue a été potentiellement introduit. Pour cela, une méthode de vérification est sélectionnée. Certaines sont plus coûteuse, mais permettent plus de sécurité quant à la découverte d'un bogue sur l'exemple.

Ces méthodes de vérifications sont les suivantes :

- vérifier toutes les solutions : comparer l'ensemble de solutions du modèle non muté avec l'ensemble de solutions du modèle muté, si ceux-ci sont différents, il existe un bogue
- compter le nombre de solutions : comparer le nombre de solutions des deux modèles, si le nombre diffère, il y a un bogue, un nombre égal n'impliquant pas forcément une absence de bogue
- vérifier une solution spécifique : sélectionner une solution du modèle non muté et vérifier qu'elle est toujours acceptée par le modèle muté
- vérifier la satisfaction : vérifier que le problème est toujours satisfaisable
- vérifier l'optimum : vérifier que l'optimum est toujours le même

3 Discussion sur les résultats et conclusion

Nous avons fait tourner *Hurricane* sur le système de modélisation CPMpy et avons découvert un nombre significatif de bogues, appartenant à plusieurs catégories et non détectés par les tests unitaires. Certains causaient des arrêts du systèmes, tandis que d'autres modifiaient l'ensemble de solutions. Notre utilisation a montré également une augmentation de la couverture de test du système. Notre méthode est également particulièrement bien conceptualisée pour de l'intégration continue, et permet de découvrir de nouveaux bogues au fur et à mesure du développement. Cependant, bien que notre méthode soit efficace, l'une des principales difficultés est la potentielle découverte d'un même bogue via plusieurs modèles mutés. Il est en effet difficile, sans analyse humaine, de détecter si la source du bogue est la même.

Notre logiciel cadre est également très modulable et peut évoluer en fonction des besoins du logiciel à tester. Il est en effet facile d'ajouter de nouveaux modèles de base, ainsi que de nouvelles mutations. Pour les systèmes de modélisation intégrant plusieurs solveurs cibles, une nouvelle méthode de vérification simple peut être de résoudre le modèle muté avec plusieurs solveurs et voir si ceux-ci s'accordent sur la vérification. Nous pensons donc qu'il peut être utile pour tester tout système de modélisation.

Remerciements

Ce projet a reçu du financement du Conseil Européen de la Recherche (ERC) sous le programme de recherche et d'innovation Horizon 2020 (No. 101002802, CHATOpt).

Références

[1] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[2] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20 : 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020.

[3] Wout Vanroose, Ignace Bleukx, et al. Mutational fuzz testing for constraint modeling systems. In *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307, pages 29–1. Schloss Dagstuhl–Leibniz-Zentrum, 2024.