

Résumé

L'objectif de ce premier TP est de prendre en main l'API Java du solveur Z3 et de résoudre un premier petit problème pouvant être exprimé sous forme d'un problème SAT.

1 Utilisation de Z3 comme un solveur SAT

Nous allons dans la suite utiliser le solveur SMT Z3 [1], un démonstrateur de théorèmes développé par Microsoft Research. Nous allons dans un premier temps l'utiliser comme un solveur SAT, i.e. sans théorie du premier ordre. Nous utiliserons l'API Java de Z3 dont la documentation est disponible à l'URL <https://www.tofgarion.net/z3/api/html>. Toutes les classes de l'API Java de Z3 appartiennent au paquetage `com.microsoft.z3`.

1.1 Configuration de votre environnement de développement

Un exemple d'utilisation de l'API est fourni via la classe `SimpleBooleanProblem` que vous avez à disposition (la classe fait partie du paquetage `fr.n7.sat`). Un Makefile est fourni pour faciliter la compilation et l'exécution du programme. La version 4.11.2 de Z3 est normalement disponible sous `/mnt/n7fs/ens/tp_dupont/z3/`, le Makefile devrait fonctionner sans problème.

Si vous souhaitez utiliser Eclipse, VSCode ou Codium pour développer vos programmes Java utilisant Z3, ou utiliser votre machine personnelle, utilisez les instructions disponibles sur la page [configuration de Z3 pour une utilisation sur vos machines](#) sur Moodle. Ces instructions sont également valables pour l'installation disponible à l'ENSEEIH sous `/mnt/n7fs/ens/tp_dupont/z3/`.

1.2 Accéder à des informations sur Z3

Les premières lignes du programme permettent d'accéder à des informations sur Z3 (version etc.) :

```
System.out.println("* Printing Z3 infos...");

com.microsoft.z3.Global.ToggleWarningMessages(true);
Log.open("test.log");

System.out.print("Z3 Major Version: ");
System.out.println(Version.getMajor());
System.out.print("Z3 Full Version: ");
System.out.println(Version.getString());
System.out.print("Z3 Full Version String: ");
System.out.println(Version.getFullVersion());
```

1.3 Créer des variables booléennes

Pour pouvoir interagir avec le solveur, il faut manipuler deux objets :

- un objet de type `Context` qui contiendra le contexte du solveur (comme son nom l'indique). On le crée ici avec les lignes suivantes :

```
HashMap<String, String> cfg = new HashMap<String, String>();
cfg.put("model", "true");
cfg.put("proof", "true");

Context context = new Context(cfg);
```

`cfg` est une table de correspondance permettant de définir les options utilisées. Ici on choisit de générer les modèles lorsque le problème est SAT et une preuve d'insatisfiabilité lorsqu'il est UNSAT.

- un objet de type `Solver` représentant une instance du solveur.

On utilise le contexte pour pouvoir créer des *expressions booléennes* qui seront ici de simples variables booléennes :

```
BoolExpr a = context.mkBoolConst("a_name");
BoolExpr b = context.mkBoolConst("b");
BoolExpr c = context.mkBoolConst("c");
```

Attention, la chaîne de caractères passée en paramètre de `mkBoolConst` est le nom « interne » de la variable booléenne pour Z3. La variable Java peut porter un nom différent. On le voit ici :

```
System.out.println(" Printing a : " + a +
    " (notice the name is different than the variable name)");
```

On peut vérifier que la variable Java `a` représentant la variable booléenne Z3 correspondant n'a pas de valeur :

```
System.out.println(" Has a value true? " + a.isTrue());
System.out.println(" Has a value false? " + a.isFalse());
System.out.println(" Value of a? " + a.getBoolValue());
```

1.4 Créer des formules

Pour créer des formules, on utilise encore une fois le contexte. Par exemple, si l'on veut créer $a \rightarrow b$ et $\neg b \vee \neg c$:

```
BoolExpr formula1 = context.mkImplies(a, b);
BoolExpr formula2 = context.mkOr(context.mkNot(b),
    context.mkNot(c));
```

On peut remarquer que `mkOr`, comme `mkAnd`, a un paramètre variadique, donc on peut lui passer autant d'arguments que l'on souhaite ou un tableau contenant des instances de `BoolExpr`.

1.5 Vérifier si un ensemble de formules est satisfaisable

Pour vérifier qu'un ensemble de formules est satisfaisable, il suffit dans un premier temps de les ajouter au solveur :

```
BoolExpr set1[] = { formula1, formula2, a };

for (BoolExpr e : set1)
    solver.add(e);
```

Ensuite, il suffit d'appeler la méthode `check` sur le solveur et de vérifier le résultat (ces instructions sont encapsulées dans la méthode `checkAndPrint` dans l'exemple fourni) :

```
Status q = solver.check();

switch (q) {
case UNKNOWN:
    System.out.println(" Unknown because:\n" +
        solver.getReasonUnknown());
    break;
case SATISFIABLE:
    System.out.println(" SAT, model:\n" +
        solver.getModel());
    break;
case UNSATISFIABLE:
    System.out.println(" UNSAT, proof:\n" +
        solver.getProof());
    break;
}
```

Ici, l'ensemble $\{a \rightarrow b, \neg b \vee \neg c, a\}$ est satisfaisable. Un modèle est affiché au format SMT-LIB [2]. De la même façon, si l'ensemble n'était pas satisfaisable, une preuve d'incohérence serait affichée au même format.

1.6 Sauvegarder l'état du solver

On peut stocker l'état du solver à travers un mécanisme de pile. Par exemple, pour sauvegarder l'état courant du solver :

```
solver.push();
```

On peut ensuite ajouter une nouvelle formule (*c* ici), vérifier la satisfiabilité du problème et revenir à l'état précédent :

```
solver.add(c);
checkAndPrint(solver);
solver.pop();
```

1.7 Récupérer un modèle propositionnel

Si le problème est SAT, on peut récupérer un modèle propositionnel satisfaisant l'ensemble des formules dans le solver :

```
Model m = check(solver);
```

avec la méthode `check` définie comme suit :

```
static Model check(Solver solver) {
    if (solver.check() == Status.SATISFIABLE) {
        return solver.getModel();
    } else {
        return null;
    }
}
```

Une bonne pratique est de n'appeler `getModel` que si `check` a bien renvoyé `Status.SATISFIABLE`.

On peut ensuite obtenir l'interprétation des variables booléennes via le modèle :

```
System.out.println(" value of a: " + m.getConstInterp(a));
System.out.println(" value of b: " + m.getConstInterp(b));
System.out.println(" value of c: " + m.getConstInterp(c));
```

`getConstInterp` renvoie une instance de `Expr` sur laquelle on peut appliquer différentes méthodes, comme `isTrue` ou `isFalse` dans le cas d'expressions booléennes :

```
if (m.getConstInterp(a).isTrue()) {
    System.out.println(" m.getConstInterp(a).isTrue() returns " +
        " the Java boolean constant true");
}
```

Attention, la variable `a` elle-même n'a pas de valeur booléenne :

```
System.out.println(" BEWARE: a.getBoolValue() = " +
    a.getBoolValue());

System.out.println(" BEWARE: a.isTrue() = " +
    a.isTrue());

System.out.println(" BEWARE: a.isFalse() = " +
    a.isFalse());
```

2 Résolution de grilles de Sudoku

Nous allons utiliser Z3 pour résoudre un problème classique : compléter une grille de Sudoku. Un exemple de grille pouvant être utilisée est donné sur la figure 1.

Une grille de Sudoku construite à partir d'un $n > 0$ donné a les propriétés suivantes :

- la grille est de taille $n^2 \times n^2$
- les valeurs pouvant être placées dans la grille appartiennent à $\{1, \dots, n^2\}$

						8	5	1
			1		2		9	4
					8			
				3		6	4	
5			6				7	3
6								
	4	3	7	2				
	2	9	8			3		

FIGURE 1 – Grille Sudoku du journal Le Monde du 8 novembre 2016 (grille élaborée par Yan Georget)

- chaque case doit avoir une valeur
- chaque valeur doit apparaître une et une seule fois dans une ligne
- chaque valeur doit apparaître une et une seule fois dans une colonne
- chaque valeur doit apparaître une et une seule fois dans les sous-grilles de taille $n \times n$ (on parle de sous-grilles ne se recouvrant pas)

Vous trouverez dans l'archive fournie un squelette pour la classe `Sudoku` à développer. En particulier, vous disposez du constructeur et des méthodes suivantes :

- un constructeur `Sudoku` permettant de construire un problème
- une méthode `print` permettant d'afficher le modèle du solver s'il existe (affichage de la grille solution). S'il n'y a pas de valeur trouvée pour une case, le symbole `_` s'affiche. S'il y a plusieurs valeurs pour une case, la première valeur s'affichera en vidéo inverse, par exemple **1**.
- une méthode `solve` permettant de résoudre le problème
- une méthode `addValue` permettant de fixer la valeur d'une case
- une méthode statique `loadSudoku` permettant de charger un fichier au format CSV représentant une grille
- une méthode statique `représentant` un programme lisant une grille depuis un fichier et l'affichant

Des commentaires `// TODO` sont situés aux endroits que vous devez compléter.

1. lire le code et comprendre les méthodes existantes.
2. lancer le programme avec

```
make run-sudoku SUDOKU_FILE=./files/le-monde.csv
```

Que pensez-vous de la solution fournie ?

3. ajouter les formules logiques permettant de spécifier les règles du jeu dans les méthodes `addExistenceconstraints`, `addColumnconstraints`, `addRowconstraints` et `addSubgridsconstraints`.
4. vérifier sur la grille représentée sur la figure 1 que le solver trouve une solution.
5. regarder le temps nécessaire pour créer les contraintes et le temps nécessaires pour résoudre le problème. Que constatez-vous ?
6. le fichier `exemple multiple-sol.csv` contient une grille ayant plusieurs solutions (ce n'est donc pas une vraie grille de Sudoku). Comment faire pour toutes les énumérer ?

Références

- [1] MICROSOFT RESEARCH. *The Z3 theorem prover*. 2024. URL : <https://github.com/Z3Prover/>.
- [2] THE SMT-LIB INITIATIVE. *SMT-LIB – The Satisfiability Modulo Theories Library*. 2024. URL : <https://smt-lib.org/>.

License CC BY-NC-SA 3.0



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmute) and to Remix (adapt) this work under the following conditions:



Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Noncommercial – You may not use this work for commercial purposes.



Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See <http://creativecommons.org/licenses/by-nc-sa/3.0/> for more details.