`ltl2ba`
Un compilateur de formules LTL en automate de Büchi

Emile ROLLEY

8 mai 2022

Abstract

Rapide compte rendu du devoir consistant à implémenter un algorithme pour la compilation de formules LTL en automates de Büchi sur les transitions.

Introduction

L'objectif de ce devoir est d'implémenter l'algorithme[1] permettant de compiler une formule LTL en un automate de Büchi généralisé sur les transitions équivalent. Le projet prends donc la forme d'un compilateur permettant dans un premier temps de parser une formule LTL et dans un second temps de générer l'automate de Büchi équivalent – voir la Figure 1.

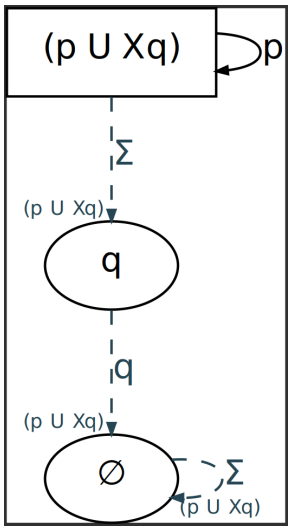

Figure 1: Exemple d'automate généré pour la formule $\varphi = p$ U Xq. La boite correspond à l'état initial et les transitions en pointillées aux conditions d'acceptations.

[1] L'algorithme est décrit dans les notes du cours d'*Introduction à la vérification* de Marc Zeitoun, et est retranscrit dans la documentation du programme : https://emilerolley.github.io/ltl2ba/ltl2ba/index.html#algo

Architecture

Le code source du projet est disponible dans le dépôt GitHub `ltl2ba`[2].

Le programme est implémenté en OCaml, en effet, ce langage est tout particulièrement efficace pour l'implémentation de compilateurs ou d'analyseurs.

Les sources sont divisées en trois parties (répertoires) :

- `src/` contient le point d'entrée du programme ainsi que le module `Cli`,
- `src/core/` contient l'ensemble des structures de données et fonctions utilisées pour l'implémentation de l'algorithme,
- `src/parsing` contient l'implémentation du parser et du lexer.

Parsing

Le parser de formule LTL est implémenté avec `ocamllex` et `menhir`. Il permet de reconnaitre la grammaire décrite dans la Figure 2. Par exemple, la formule LTL $\mathrm{G}((p \vee q) \Rightarrow \neg p)$ devra s'écrire $\mathrm{G}((p \mid q) =>!p)$.

Génération

La documentation des modules utilisés pour l'implémentation de l'algorithme est disponible ici[3].

Après la phase de parsing vient celle de la construction de l'automate.

Pour cela j'ai utilisé le module `ocamlgraph` et ai défini l'automate de Büchi généralisé sur les transitions comme un graphe orienté et étiqueté (`Core.Automata.TransBuchi`) dans lequel un nœud représente un état et est étiqueté par son ensemble de formules (`Core.Automata.FormulaSet.t`), ; les transitions sont des arcs étiquetés par l'ensemble de propositions atomiques Σ_p correspondant, de plus, s'il fait parti d'une condition d'acceptation elle est également étiquetée par les formules LTL correspondantes. Le graphe peut être sauvegardé au format Dot grâce au module `Core.Automata.TransBuchiDotPrinter`.

En ce qui concerne l'implémentation de l'algorithme, j'ai fait le choix de ne pas construire de graphe intermédiaire pour chaque état. La fonction `Core.Algorithm.red {phi}` calcule donc à la fois les états accessible depuis `phi` sans utiliser de transitions marquées et pour chaque α, l'ensemble des états accessibles en empruntant uniquement des transitions marquées par α. Ces états son stockées dans la structure de données `Core.Algorithm.red_states`.

Finalement, les états et transitions de l'instance du `TransBuchi` sont ajoutés *à la volée* (dans le fichier `src/ltl2ba.ml`) en appliquant récursivement `Core.Algorithm.red` sur l'ensemble des états générés.

[2] Adresse du dépôt : https://github.com/EmileRolley/ltl2ba

[3] La documentation est disponible à l'adresse suivante : https://emilerolley.github.io/ltl2ba/index.html

$\varphi ::=$	formule LTL
$\quad \mid$ `false` $\mid$ `true`	booléens
$\quad \mid \rho$	proposition atomique
$\quad \mid \circledcirc\, \varphi$	opérateur unaire
$\quad \mid \varphi \circledast \varphi$	opérateur binaire
$\quad \mid$ `(`φ`)`	formule parenthésée
$\circledcirc ::=$ `!`	négation booléenne
$\quad \mid$ `X`	opérateur temporel *Next*
$\quad \mid$ `F`	opérateur temporel *Finally*
$\quad \mid$ `G`	opérateur temporel *Globally*
$\circledast ::=$ `|`	disjonction booléenne
$\quad \mid$ `&`	conjonction booléenne
$\quad \mid$ `=>`	implication booléenne
$\quad \mid$ `U`	opérateur temporel *Until*
$\quad \mid$ `R`	opérateur temporel *Release*

$\rho ::= [a-z][a-zA-Z0-9]^*$

Figure 2: Grammaire des formules LTL reconnaissables.

Fonctionnalités

Dans sa dernière version, `ltl2ba.0.2.0` est capable de parser n'importe qu'elle formule LTL respectant la grammaire décrite dans la Figure 2. Pour cela, après avoir compilé l'exécutable en suivant les instructions décrites dans le `README`, il suffit de spécifier en argument la formule LTL à parser – voir la Figure 3.

```
» ltl2ba "G(p ⟹ FXq)"
--- ltl2ba v0.2.0 ---
[OK]      φ := G(p ⇒ FXq)
[OK] NNF(φ) := (⊥ R (¬p v (⊤ U Xq)))
```

Figure 3: Exemple d'exécution pour la formule $\mathrm{G}(p \Rightarrow \mathrm{XF}q)$.

Afin de sauvegarder l'automate généré au format dot, il suffit de préciser le chemin vers le fichier souhaité grâce à l'option `-d` – voir la Figure 4 et 5.

```
» ltl2ba "G(p ⟹ FXq)" -d graph.dot
--- ltl2ba v0.2.0 ---
[OK]      φ := G(p ⇒ FXq)
[OK] NNF(φ) := (⊥ R (¬p v (⊤ U Xq)))
[OK] Automata successfully printed in 'graph.dot'
```

Figure 4: Exemple d'exécution avec l'option `-d`

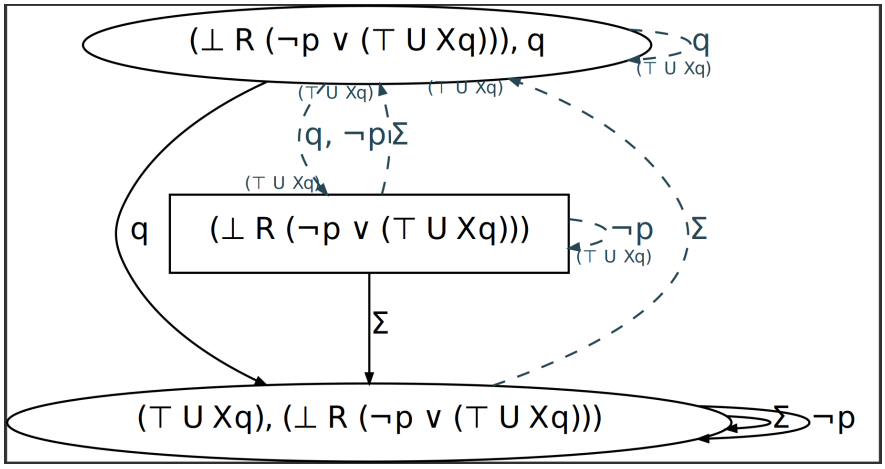

Figure 5: Résultat de la commande `xdot graph.dot`

L'état initial de l'automate est rectangulaire et les transitions acceptantes sont en pointillées et coloriées avec une couleur associée à la formule LTL de la condition d'acceptation correspondante. Lorsqu'une transition appartient à plusieurs conditions d'acceptations alors est coloriée avec la moyenne des couleurs de chacune des conditions – voir la Figure 6.

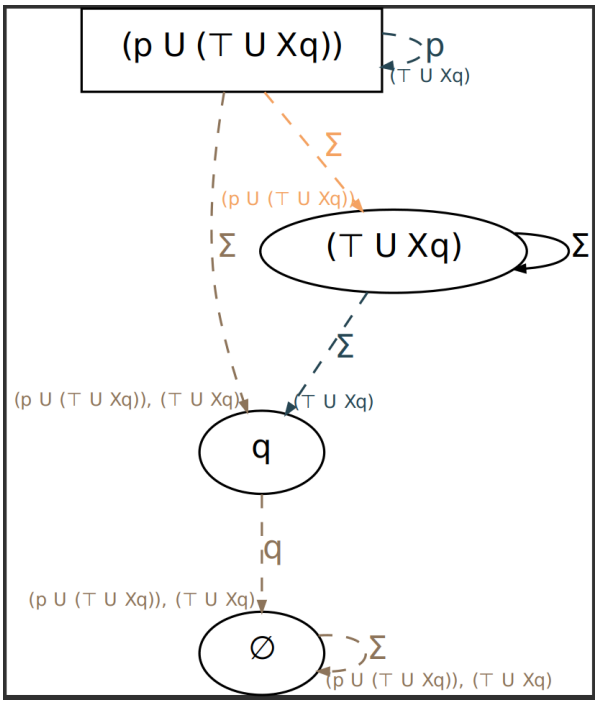

Figure 6: Exemple d'automate avec des transitions appartenant à plusieurs conditions d'acceptations

Il également possible d'afficher plus d'information lors de l'exécution en précisant l'option `-v` – voir la Figure 7.

```
» ltl2ba "p | Fq" -v
--- ltl2ba v0.2.0 ---
[LOG] Parsing formula...
[OK]      φ := (p ∨ Fq)
[LOG] Calculating NNF...
[OK] NNF(φ) := (p ∨ (⊤ U q))
[LOG] Translating to automata...
[LOG]   Y = {(p ∨ (⊤ U q))}
[LOG]   Reducing state: (p ∨ (⊤ U q))
[LOG]   Reducing state: (⊤ U q)
[LOG]           - Adding acceptance transition for ((⊤ U q)): (p ∨ (⊤ U q)) -- p → ∅
[LOG]           - Adding acceptance transition for ((⊤ U q)): (p ∨ (⊤ U q)) -- p, q → ∅
[LOG]           - Adding transition: (p ∨ (⊤ U q)) -- Σ → (⊤ U q)
[LOG]   Y = {∅}
[LOG]           - Adding acceptance transition for ((⊤ U q)): ∅ -- Σ → ∅
[LOG]   Y = {(⊤ U q)}
[LOG]   Reducing state: (⊤ U q)
[LOG]           - Adding acceptance transition for ((⊤ U q)): (⊤ U q) -- q → ∅
[LOG]           - Adding transition: (⊤ U q) -- Σ → (⊤ U q)
```

Figure 7: Exemple d'exécution avec l'option `-v`

Conclusion

Avoir fait le choix de ne pas construire le graphe intermédiaire à engendré de nombreuses difficultés, notamment sur la gestion des conditions acceptantes, et à forcer l'usage de structure de données et étapes de calculs supplémentaires complexifiant le code pour un gain de performance non mesuré. Je pense qu'il serait intéressant d'avoir une implémentation vraiment fidèle à l'algorithme afin de comparer si un gain de performance est observable, ainsi que de comparer la compréhension du code.

Malgré cela, j'ai réussi à implémenté une version fonctionnelle. De plus, j'ai beaucoup appris sur la gestion de projet en OCaml, en effet, j'ai pu mettre en place de l'intégration continue avec GitHub : exécution automatique des tests unitaires et de leurs couvertures, ainsi que la mise en ligne de la documentation – pour la quelle j'ai porté une attention toute particulière.