Le model-checking

Arabella BRAYER
Université libre de Bruxelles

Abstract

La vérification de modèle est sujet à beaucoup d'intérêts dans le paysage informatique actuel. Pour s'en convaincre, il suffit de regarder le nombre d'applications en développement ou de sujets de recherche. Dans ce travail, le principe est expliqué, puis une application particulière : Prism, est présentée. Pour ce faire, ce document s'appuie sur le problème du dîner des philosophes, modélisé en processus de décision markovien.

Introduction

Dans le monde informatique actuel, les procédures de développement d'un projet sont extrêmement variées. Il y a beaucoup de façons différentes de suivre un cycle de développement. Toutefois, dans chacune de ces procédures revient un thème particulièrement important : la correction. Les professionnels accordent du temps et des budgets pour que l'étape de vérification soit correctement réalisée. Il existe des procédures différentes afin de s'assurer que le projet répond bien au problème posé en toute sécurité. Celles-ci seront plus ou moins adaptées en fonction du projet.
La **vérification de modèle** (model-checking) est une façon de prouver qu'un programme est correct. Elle est une étape à part entière du développement de certains types de projets, par exemple utilisant des appareils en temps réel [Pagetti (2014)].
Le document présent a pour objectif de présenter l'un des outils de model-checking : **Prism**, en s'appuyant sur l'exemple du **dîner des philosophes**. Celui-ci est modélisé en **processus de décision de Markov** (Markov Decision Process, MDP).

Prism : un logiciel de model-checking

La vérification de modèle

Les tests font désormais partie du développement normal d'une application. Il y en a de plusieurs sortes, comme les tests unitaires, ou d'intégration, mais globalement, la façon la plus habituelle de vérifier le fonctionnement du programme consiste à implémenter des tests vérifiant l'exécution dans des conditions prévues à l'avance. Cependant, cette façon de procéder laisse une place très importante à des erreurs. De plus, de trop nombreux accidents ont montré que le manque de vérifications pouvait avoir des conséquences dramatiques, comme par exemple le célèbre incident impliquant la fusée Ariane 5 [Wikipedia (2016)], ou des pacemakers défaillants en raison de leur programme [Sandler (2010)].

Pour certaines applications qui doivent fonctionner dans des conditions où la sécurité est importante (avions), ou en temps réel (appareils de détection, canaux de communication, etc.), une façon de mettre à l'épreuve le projet consiste à abstraire la solution en la modélisant. Une fois l'abstraction faite et le modèle ainsi obtenu, il devient possible d'utiliser un outil automatique et systématique comme l'ordinateur afin d'analyser ce modèle de plusieurs façons : correction, efficacité, etc.
Cette étape intervient globalement au début de la réalisation, avant l'implémentation en elle-même, avant d'autres éventuels tests.

Prism

À l'heure actuelle, il existe toute une gamme d'applications destinées au model-checking, comme par exemple Prism [Dave Parker (2016)], JPF [NASA (2016)], ou Blast [Berkeley (2014)]. Ces applications diffèrent de plusieurs façons : elles sont plus ou moins libres, et permettent de traiter certains types de modèles informatiques. Il existe une multitude de modèles, aux propriétés différentes. Prism s'est spécialisé dans l'analyse de certains *modèles stochastiques*.

Qu'est-ce qu'un modèle stochastique ? Un modèle stochastique est une représentation abstraite d'un processus observé auquel est attribué un caractère aléatoire. Cela peut être un modèle biologique, de météo, la transmission dans un canal de communication, ou encore beaucoup d'autres exemples. Ces modèles sont très nombreux. Leur nombre s'explique par les nuances dans l'objet qu'ils décrivent. Ceux peuvent en effet être à temps discret ou continu, leurs variables peuvent également être discrètes ou continues, ainsi que les éléments qui les constituent. Par exemple, les modèles de Kripke ou de Markov vont permettre de

modéliser des environnements différents.

Un logiciel de vérification de programme aura implémenté des algorithmes afin d'effectuer la tâche d'analyse sur le modèle encodé par l'utilisateur. Comme ces modèles sont très nombreux, tous ne seront pas acceptés par une même application.
PRISM est un logiciel libre de cette famille, s'étant spécialisé dans les modèles stochastiques de type Markovien. Il est développé par une équipe d'universitaires (Dave Parker, Gethin Norman, Marta Kwiatkowska), ayant largement documenté leur produit et le faisant régulièrement évoluer. Il est par conséquent un outil fort utile à connaître.

Les modèles gérés par Prism sont au nombre de cinq :
- Discrete-time Markov Chains (DTMCs)
- Continuous-time Markov chains (CTMCs)
- Markov decision processes (MDPs)
- Probalistic temporal logics (PAs)
- probabilistic timed automata (PTAs)

Prism possède son propre langage qui permet de les implémenter, et propose des outils de vérification et d'analyse spécifiques à chaque type de modèle. Dans la suite de ce document, nous nous intéresserons principalement aux MDPs.

Le problème des philosophes

5 philosophes sont autour d'une table, sur laquelle reposent 5 assiettes de pâtes, et 5 fourchettes entre chaque assiette. Ces philosophes peuvent être dans trois situations : penser, avoir faim ou être en train de manger. Pour manger cependant, il leur faut une fourchette dans chaque main. Certaines situations peuvent mener à un blocage. Par exemple, si les philosophes décident simultanément de se saisir de la fourchette qui est à leur gauche, ils vont tous être dans l'attente infinie de celle de droite, sans qu'aucun philosophes ne puisse se rassasier. Quand bien même une règle autoriserait les philosophes à reposer leur fourchette passé un certain délai d'attente, cela n'éviterait pas le risque de blocage.

Le but du dîner des philosophes est de s'assurer que chacun va pouvoir manger. S'il est fréquemment employé comme métaphore du problème de l'ordonnancement des processus d'un ordinateur, c'est qu'il illustre bien la difficulté de garantir l'accès aux ressources lorsque plusieurs processus sont en exécution en même temps. Il est en effet important de s'assurer que chacun va pouvoir fonctionner correctement.
Les blocages cités plus haut sont appelés *interblocages*, ils représentent un état où plus aucun processus n'accède aux ressources. La mort d'un philosophe signifie quant à elle que les ressources ne lui sont jamais allouées. Il est admis qu'au bout d'un certain temps (arbitraire), celui-ci est considéré comme mort.

Ce problème a fait l'objet de nombreuses publications et algorithmes. Celle de Lehmann and Rabin (1981) contient un résultat important : Lehmann et Rabin y démontrent qu'il n'existe pas d'algorithme déterministe, symétrique et distribué qui évite les interblocages ni la mort d'un philosophe. Ils proposent une implémentation probabiliste de celui-ci, qui rompt avec le problème de la symétrie. Concrètement, lorsqu'un philosophe a faim, il choisit aléatoirement de prendre la fourchette droite ou gauche en premier. L'ajout de cette simple propriété permet à l'algorithme d'éviter les situation d'interblocage, mais pas de famine dans tous les cas. Cet algorithme se trouve très souvent en référence dans littérature scientifique qui aborde l'ordonnancement, ou le dîner des philosophes. Voici la version proposée en 1981 par Lehmann et Rabin :

```
while true
  do think;
    do trying:=true or die od;
    while trying
      do draw a random element s of {Right, Left}
      *** with equal probabilities ***
        wait until s chopstick is down
        and then lift it;
        if R(s) chopstick is down
          then lift it;
            trying:= false;
          else
            put down s chopstick
        fi
      od;
    eat;
    put down both chopsticks;
    *** one at a time, in arbitrary order ***
od;
```

Afin de montrer comment fonctionne Prism, c'est cet algorithme qui est modélisé ici, sous forme de MDP.

Markov decision processes (MDPs)

Définition formelle

Un MDP est défini syntaxiquement de cette façon :
un tuple $\{S, s\_init, T, Steps, L\}$ où
- S est un ensemble d'états
- s_init est l'état initial $\in S$
- T est l'espace de temps. Il est discret.
- $Steps$ est telle que $S \rightarrow 2^{A*Dist(S)}$ est une fonction de transition, où A est un ensemble d'actions, et $Dist(S)$ est l'ensemble des distributions de probabilités de S tel que $f \in Dist(S)$) :
  - $\forall s \in S : f(s) \in [0, 1]$
  - $\sum_{s \in S} f(s) = 1$

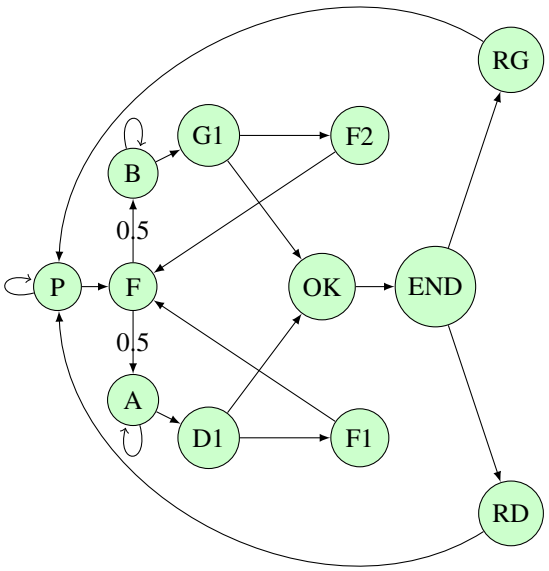
— $L : S \rightarrow 2^{AP}$ est l'ensemble des propositions atomiques, où A représente l'ensemble des actions et P les distributions de probabilités associée à chaque action.

L'espace de temps peut être fini ou infini. S et A sont de cardinalités finies.

Ce qui définit avant tout un processus de Markov est qu'il vérifie l'hypothèse de Markov. Cela signifie que ses états passés sont sans influence sur son état présent. Il n'y a pas donc pas de « mémoire », ou alors, celle-ci n'apporte pas d'information utile à la prédiction du prochain état.
Par ailleurs, un processus de Markov donné, dans un état donné, peut — par le biais d'une action — passer dans un autre état en récoltant une « récompense ». Celle-ci peut-être positive ou négative, et donc être considérée comme un coût, ou une pénalité. La différence entre une chaîne de Markov et un MDP repose sur les actions. On peut considérer qu'une chaîne de Markov est un cas particulier de MDP où il n'existerait pas plus d'une seule action par état. Dès lors qu'il y a multiplicité des actions, cela différencie une chaîne de Markov d'un processus de Markov.

Exemple d'une modélisation en MDP

Figure ***Le dîner des philosophes sous forme de MDP***



Description :

P en train de penser
F a faim
A, B tirage au sort
D1 a pris la fourchette droite si elle était libre
G1 a pris la fourchette gauche si elle était libre
F1 la fourchette gauche était prise, a reposé la droite
F2 la fourchette droite était prise, a reposé la gauche
OK possède les deux fourchettes, commence à manger
END est repu
RD a reposé la fourchette droite
RG a reposé la fourchette gauche

Le schéma ci dessus comporte ces éléments :

— $S = \{P, F, A, B, D1, G1, F1, F2, OK, END, RD, RG\}$
— $s_{init} = P$
— le temps est infini
— Les actions ne sont pas représentées ici par souci de lisibilité, cependant il faut considérer qu'une action est nécessaire entre chaque état. Plus particulièrement, deux actions sont accessibles depuis l'état « en train de penser » qui donneront accès chacune à une transition chacune, de probabilité 1. Ainsi l'état « en train de penser » aura — par l'intermédiaire d'une action nommée par exemple « continuer » — une probabilité de 1 de retourner à l'état « en train de penser ». Ce point est abordé ci-après.
— les fonctions de transition sont représentées par les arcs. Certaines sont non déterministes, comme le fait de se mettre à avoir faim, d'autres sont probabilistes, comme le fait de tirer au sort entre F et A, ou F et B.

Les philosophes sont initialement dans l'état « en train de penser », et le temps est infini. Il faut noter qu'ici, le modèle est considéré comme « équitable », car les philosophes tentent de manger uniquement quand ils ont faim. Ceci est important à noter car tous les algorithmes du dîner des philosophes ne reposent pas sur cette idée, et par conséquent, ne mènent pas exactement au même modèle.

C'est donc par l'intermédiaire d'actions que les états sont atteints. Par exemple, depuis l'état B, le philosophe atteindra l'état G1 en effectuant l'action « prendre la fourchette gauche ». Il est important de noter que pour chaque « action » est associé un ensemble de probabilités dont la somme est égale à 1. Par exemple, il y a deux actions possibles dans l'état P : continuer à penser, ou se mettre à essayer de manger (commencer à avoir faim). Ces deux actions ne sont ni déterministes, ni probabilistes. L'action « continuer de penser » ira dans l'état « penser » avec une probabilité de 1, et l'action « commencer à essayer de manger » ira dans l'état F avec une probabilité de 1 également.

Maintenant que cet exemple a permis d'illustrer la définition d'un MDP, il reste à introduire quelques termes utiles à la compréhension de ce modèle :

Un **chemin**(path) est une séquence d'états et de paires

d'action/distribution. Il peut être fini ou infini. Il représente une exécution (instance) de la modélisation. Comme un MDP n'est pas déterministe, deux exécutions pourront produire deux chemins différents.
Formellement, un chemin peut s'écrire :
$path = (S \times A \times [0,1])^*$, et un exemple de chemin issu du MDP du modèle précédent peut être :
$(P(a_0,1), F(a_1,1), B(a_0,0.5), B(a_1,1), G1(a_0,1)$

Path(s) représente l'ensemble des chemins (finis) qui débutent à l'état s.

Un **adversaire**(adversary) est un concept qui est souvent employé dans la littérature scientifique à propos des MDPs au travers du mot *politique* (policy). Plus formellement, un adversaire peut s'écrire $\sigma : Path(s) -> a$, où a est l'action effectuée, c'est donc une fonction σ qui peut générer tous les chemins finis $Path(s)$ qui font l'action a_i à une position donnée.

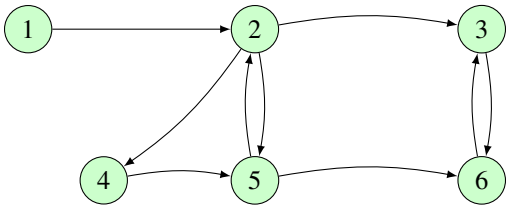
Dans le modèle du dîner des philosophes, l'état « penser » dure un temps indéterminé, non stochastique. Par conséquent, en l'état, il n'est pas possible de raisonner à l'aide des probabilités sur ce modèle. C'est ici qu'interviennent les adversaires, en permettant de résoudre un choix non déterministe, en fixant les choix. Le modèle ainsi obtenu ne possède plus d'actions multiples sur ses états, et peut être considéré comme un DTMC. Or, les DTMC permettent de raisonner à l'aide des probabilités. Il devient possible de déduire des fourchettes de probabilités minimum et maximum d'atteindre chaque état du MDP, en se basant sur des adversaires différents.
Par exemple, dans le dîner des philosophes, si l'action « continuer » est faite un nombre infini de fois, le philosophe ne passera jamais dans l'état « a faim », alors que dès que l'action « commencer à essayer de manger » aura été effectuée, la probabilité de manger — calculée depuis le DTMC obtenu — sera de 1.
Concrètement, toujours avec l'exemple du dîner des philosophes et en le tronquant, les premiers adversaires peuvent être σ_n, où n indique à quelle étape le choix « commencer à avoir faim » est fait. σ_1 serait l'ensemble des chemins $Path(P)$ tels que F arrive en seconde position :
$(P, a_0), (F, a_1, ...)$,
σ_2, $Path(P)$ tels que F arrive en troisième position :
$(P, a_0), (P, a_0), (F, a_1), ...$,
et plus généralement, σ_n, l'ensemble des $Path(P)$ où le choix F arrive en position $n+1$: $\underbrace{(P, a_0)}_{nfois}, ..., (F, a_1), ...$ De ces ensembles de chemins se déduisent des DTMCs.

DTMC (Discrete Time Markov Chain)

Un **DTMC** est — comme son nom l'indique — une chaîne de Markov. Les propriétés sont proches du MDP, mais à la différence de celui-ci, il n'y a pas d'actions précédent le changement d'état, il n'y a donc qu'une seule distribution de probabilité sur chaque état. Le DTMC respecte aussi la propriété de Markov, à savoir que la probabilité de son changement d'état ne dépend que de son état courant, pas de son passé.
Un DTMC est souvent représenté par un graphe dont les nœuds représentent les états, et les arcs les transitions d'état. Le poids des arcs représente la probabilité de la transition d'un état à un autre. La somme de tous les arcs sortants de chaque état est égale à 1. Un DTMC peut posséder des **SCC** (*Strongly Connected Components* : Composants fortement connexes).

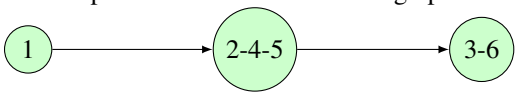
SCC et BSCC ((Bottom) Strongly Connected Components)

Dans un graphe Γ, composé d'états S et d'arcs dont le poids représente la probabilité d'atteindre l'état s' depuis l'état s, un SCC T est un sous ensemble d'états de S si pour chaque paire d'états s et s' de T, il existe un chemin de s à s'.
Voici un exemple de SCC. Pour plus de clarté, les probabilités n'ont pas été indiquées :



Dans cet exemple ressortent plusieurs SCCs. Par exemple, L'ensemble d'états {2,4,5} en forme un, car pour atteindre 2, il faut être dans l'état 1, ou 5 ; Pour atteindre 4, dans l'état 2. L'état 5 requiert de provenir de l'état 2. On voit donc apparaître un cycle qui se forme avec les trois seuls états {2,4,5} dans lequel 1 n'apparaît pas car on ne peut plus atteindre depuis les autres états.

Il est possible de résumer ce graphe en celui-ci :



Il existe également une structure appelée **BSCC**, qui est un SCC dont aucun état en dehors de T n'est atteignable. Une seule de ces structures apparaît dans le graphe Γ, c'est le couple d'états {3,6}. Une fois l'un de ces deux états atteint, il ne sera plus possible d'en atteindre d'autres.

Les MDPs possèdent également des SCCs et BSCCs. Ces structures permettent de raisonner de façon plus globale avec les probabilités.

Méthodes de résolution

Prism implémente des algorithmes capables de générer depuis des MDPs des probabilités minimales et maximales d'atteindre certains états, depuis un état d'origine.
Formellement, si $Prob^{\sigma}(s,\alpha)$ est la probabilité dans l'adversaire σ d'atteindre l'état s par l'intermédiaire de l'action α, ceci peut s'écrire de cette façon :
$p_{min}(s,\alpha) = inf_{\sigma \in Adv} Prob^{\sigma}(s,\alpha)$ et
$p_{max}(s,\alpha) = sup_{\sigma \in Adv} Prob^{\sigma}(s,\alpha)$
où σ représente un adversaire, Adv représente l'ensemble de tous les adversaires, s représente l'état initial, et α un état à atteindre.
Il y a principalement trois méthodes pour calculer les probabilités minimum et maximum dans un MDP :

- Le point fixe (appelée méthode itérative dans Prism)
- La programmation linéaire
- L'itération de politique

En pratique, Prism emploie la première méthode par défaut, mais il est possible de le configurer afin qu'il emploie les deux autres. La seconde offre de bonnes performances mais sur des petits exemples. Les deux premières sont présentées dans ce document. Ce sont des méthodes *numériques*, ce qui signifie que ce n'est pas par l'analyse que le résultat sera obtenu, mais par itération d'un procédé permettant d'accéder à une approximation du résultat.

Le point fixe

La méthode du point fixe est une procédure éprouvée dans les méthodes algébriques. Elle consiste à résoudre $x = f(x)$ [Quarteroni et al. (2007)] . Toutes les fonctions ne sont pas éligibles à l'emploi de cette méthode, mais concernant le calcul des probabilités dans un DTMC, il peut être montré que c'est le cas.

Les explications suivantes sont issues des lectures proposées sur le site du logiciel, notamment de celle-ci : Parker (2011b).
Il peut donc être montré que pour les probabilités minimum d'atteindre α :
$p_{min}(s,\alpha) = lim_{n\to\infty} x_s^{(n)}$ où $x_s^{(n)}$ vaut :

- 1 si $s \in Sat(\alpha)$ (l'état α est atteint)
- 0 si $s \in S^{min=0}$ (un autre état est atteint)
- 0 si $s \in S^{?}$ et $n = 0$ (sur un état autre que α dans l'itération n=0)
- $\{min \sum_{s' \in S} \mu(s') \cdot x_{s'}^{n-1} | (\alpha,\mu) \in Steps(s)\}$ si $s \in S^{?}$ et $n > 0$ (sur les états intermédiaires)

où $S^{?} = S \backslash (Sat(\alpha) \cup S^{min=0})$ c'est à dire les états intermédiaires, et la méthode s'arrête quand la solution converge suffisamment, c'est à dire lorsque :
$| \ x^{n-1} - x^{n} \ |) < \epsilon$ avec ϵ jugé suffisamment petit pour garantir une approximation satisfaisante du résultat. Il s'agit là de l'équation de Bellman, notée ici $F(x^n)$. La méthode peut se résumer alors de cette façon :

$x^0 = 0 \forall s \in S \setminus Sat(a)$
$x^{n+1} = F(x^n)$
$p_{min}(a) = \lim_{n\to\infty} x^n$

Le calcul de la probabilité maximum est similaire à la différence que la fonction itérée n'est plus la fonction *min* mais *max*.

Recherche d'accessibilité

Une première étape importante de l'algorithme consiste à étudier l'accessibilité des états, afin de pouvoir facilement distinguer ceux non-accessibles ($S^{min=0}$ ou $S^{max=0}$).

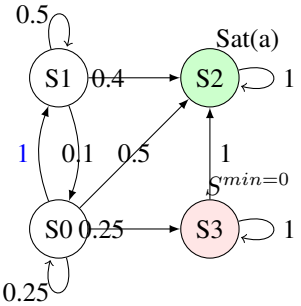
Data: R ← Sat(a) ;
Done ← false ;
while *done = false* **do**
  $R' = R \cup \{s \in S | \exists (a,\mu) \in Steps(s) t.q. \exists s' \in R t.q. \mu(s') > 0\}$;
  if $R' = R$ **then**
    done ← true ;
  end
  $R \leftarrow R'$;
end
return $S \backslash R$;

Algorithm 1: Recherche d'accessibilité

L'algorithme décrit la méthode pour rechercher l'ensemble $S^{max=0}$. Une très légère différence existe pour trouver $S^{min=0}$. Dans ce cas, la recherche de R dans la boucle while se fait de cette façon : $R' = R \cup \{s \in S | \forall (a,\mu) \in Steps(s) \times \exists s' \in R \times \mu(s') > 0\}$.

Exemple avec le point fixe

Il est plus facile de s'appuyer sur un exemple afin de comprendre comment le calcul se déroule. Le dîner des philosophes étant un modèle trop complexe, c'est un modèle plus simple, issu de la documentation qui sera exploité ici.



Voici le détail d'une exécution de la méthode du point fixe sur cet exemple. Dans un premier temps, à l'aide de l'algorithme d'accessibilité, l'état $S3$ est considéré comme appartenant à $S^{min=0}$. Lors de la première itération de l'algorithme, un vecteur est construit, représentant la probabilité d'atteindre l'état $S2$ depuis chaque état $s_i \in S$:

$[x_0, x_1, x_2, x_3]$. Les états $S0$ et $S1$ étant transitoires, x_0 et x_1 prennent la valeur 0. $S2$ est sur l'état à atteindre : x_2 prend la valeur 1, et $S3$ est dans $S^{min=0}$, x_3 vaut 0. Le vecteur prend ces valeurs : $[0, 0, 1, 0]$.
Lors de l'itération 1, seuls les états transitoires vont être réévalués, en appliquant la formule citée précédemment : $\{min \sum_{s' \in S} \mu(s') \cdot x_{s'}^{n-1} | (\alpha, \mu) \in Steps(s)\}$. C'est à dire pour x_0 :
$\min(1 \times 0; 0.5 \times 1 + 0.25 \times 0 + 0.25 \times 0)$,
et pour x_1 :
$\min(0.5 \times 0 + 0.4 \times 1 + 0.1 \times 0)$,
Le vecteur actualisé vaut désormais [0, 0.4, 1, 0].
De la même façon, l'itération suivante (2) va calculer :
$[\min(1 \times 0.4; 0.5 \times 1 + 0.25 \times 0 + 0.25 \times 0); \min(0.5 \times 0.4 + 0.4 \times 1 + 0.1 \times 1); 1; 0]$, et ainsi de suite jusqu'à converger. Le vecteur final sera une approximation de $[\frac{2}{3}; \frac{14}{15}, 1, 0]$. Il suffit à la fin de sélectionner le meilleur adversaire. Sur S0, deux adversaires sont en concurrence, cela revient à appliquer $\min(1 \times \frac{14}{15}; 0.5 \times 1 + 0.25 \times 0 + 0.25 \times \frac{2}{3})$ (resp. max). Le résultat retourné est $\approx \frac{2}{3}$. L'approximation provient de la méthode d'arrêt : comme expliqué précédemment, lorsque la distance entre deux itérations est jugée suffisamment petite, l'algorithme s'arrête. Globalement, lorsque
$| x^{n-1} - x^n |) < \epsilon$. Par défaut, la valeur du ϵ dans Prism vaut 10^{-6} mais il est possible de le configurer autrement.

Programmation linéaire

Le second algorithme implémenté est une méthode de programmation linéaire. Comme toutes les autres méthodes du genre, elle consiste à optimiser une fonction objectif tout en respectant un certain nombre de contraintes (égalités ou inégalités). Le lecteur qui souhaiterait en savoir davantage sur ces méthodes génériques de calcul est invité à se procurer notamment cette référence : Y. De Smet (2013).
Comme pour la méthode du point fixe, l'algorithme va chercher à résoudre les valeurs des états dans $S^?$. Pour trouver la probabilité minimum, cela revient à $S \setminus (S^{min=0} \cup Sat(a))$ (et donc $S \setminus (S^{max=0} \cup Sat(a))$ pour la probabilité maximum). Puis le résultat sera obtenu en résolvant le programme linéaire suivant, dans le cas de la recherche de probabilité minimale :

Maximiser $\sum_{s \in S^?} x_s$ en respectant les contraintes
$x_s \leq \sum_{s' \in S^?} \mu(s') \times x_{s'} + \sum_{s' \in Sat(a)} \mu(s')$
$\forall s \in S^? et \forall (a, \mu) \in Steps(s)$

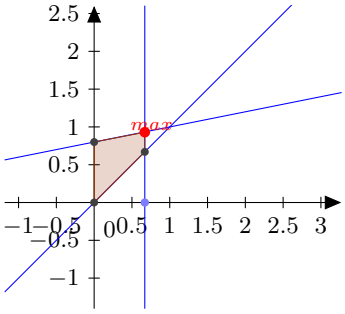
ou de la recherche de probabilité maximale :

Minimiser $\sum_{s \in S^?} x_s$ en respectant les contraintes
$x_s \geq \sum_{s' \in S^?} \mu(s') \times x_{s'} + \sum_{s' \in Sat(a)} \mu(s')$
$\forall s \in S^? et \forall (a, \mu) \in Steps(s)$

En reprenant l'exemple précédent, pour chercher la probabilité minimum, cela donne un programme de cette forme :
$\max(x_0 + x_1)$ respectant

$$\begin{cases} x_0 \leq x_1 \\ x_0 \leq 0,25 \times x_0 + 0,5 \\ x_1 \leq 0,1 \times x_0 + 0,5 \times x_1 + 0,4 \end{cases}$$

Il y a plusieurs façons de résoudre ce système. L'une d'elle est géométrique, et consiste à construire un polygone constitué de chacune de ces inéquations. Il peut être montré que la solution optimale se trouve sur l'un des sommets de ce polygone.

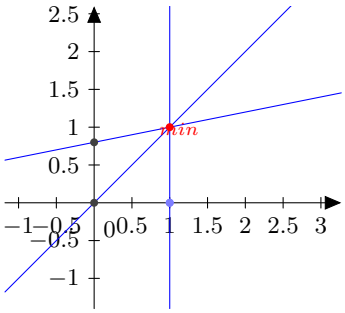


Le résultat retourné est le point $[\frac{2}{3}, \frac{14}{15}]$, la valeur en s_0 est $\frac{2}{3}$.

Afin de trouver la probabilité maximum, la méthode diffère quelque peu. De plus, l'ensemble des sommets $S_{max=0}$ est vide. Le programme est donc :
$\min(x_0 + x_1 + x_3)$ en respectant

$$\begin{cases} x_0 \geq x_1 \\ x_0 \geq 0,25 \times x_0 + 0,25 \times x_3 + 0,5 \\ x_1 \geq 0,1 \times x_0 + 0,5 \times x_1 + 0,4 \end{cases}$$

Cela revient à résoudre géométriquement :



Le résultat retourné est 1.

Ici ont été exposées deux méthodes qui offrent des performances différentes. Sur un petit exemple, la seconde est très rapide, mais elle peut devenir coûteuse sur des exemples plus grands, tandis que la première sera préférable dans ce cas [Prism (2010a)]. Les deux méthodes requièrent d'effectuer une recherche d'accessibilité au préalable.

Les explications plus haut n'ont porté que sur des instances simples. Or, Prism permet d'analyser plusieurs instances d'un même modèle en parallèle, comme ce sera le cas dans

le dîner des philosophes. Afin de procéder à l'analyse, la première étape ne sera pas la recherche d'accessibilité, mais le produit cartésien du modèle autant de fois qu'il y a d'instances. Concrètement, chaque état sera alors un vecteur représentant tous les états des instances.

Il est important de noter que Prism ne se limite pas à de l'analyse de probabilités. Il est possible de vérifier des propositions sur un modèle. Ce travail ne se concentre pas sur ce sujet, mais la façon de procéder est détaillée dans un document disponible sur le site de Prism, à la rubrique *Lectures* [Parker (2011a)] .

Étude de cas : Le dîner des philosophes

À présent que certaines méthodes numériques ont été présentées, il reste à présenter l'utilisation du logiciel en lui-même. Prism utilise son propre langage, qui est largement documenté sur le site dans la section *Manual* [Prism (2010a)]. Par ailleurs, des exemples sont présents dans le répertoire d'installation du logiciel.
Le dîner des philosophes est un problème qui est employé à des fins pédagogiques dans la section *tutoriel* [Prism (2010b)].

```
mdp

formula lfree = (p2>=0&p2<=4)|p2=6|p2=10;
formula rfree = (p3>=0&p3<=3)|p3=5|p3=7|p3=11;

module phil1

    p1: [0..11];

    [] p1=0 -> (p1'=0); // stay thinking
    [] p1=0 -> (p1'=1); // trying
    [] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly
    [] p1=2 &  lfree -> (p1'=4); // pick up left
    [] p1=2 &  !lfree -> (p1'=2); // left not free
    [] p1=3 &  rfree -> (p1'=5); // pick up right
    [] p1=3 &  !rfree -> (p1'=3); // right not free
    [] p1=4 &  rfree -> (p1'=8); // pick up right (got left)
    [] p1=4 & !rfree -> (p1'=6); // right not free (got left)
    [] p1=5 &  lfree -> (p1'=8); // pick up left (got right)
    [] p1=5 & !lfree -> (p1'=7); // left not free (got right)
    [] p1=6  -> (p1'=1); // put down left
    [] p1=7  -> (p1'=1); // put down right
    [] p1=8  -> (p1'=9); // move to eating (got forks)
    [] p1=9  -> (p1'=10); // finished eating and put down left
    [] p1=9  -> (p1'=11); // finished eating and put down right
    [] p1=10 -> (p1'=0); // put down right and return to think
    [] p1=11 -> (p1'=0); // put down left and return to think

endmodule

// construct further modules through renaming
module phil2 = phil1 [ p1=p2, p2=p3, p3=p1 ] endmodule
module phil3 = phil1 [ p1=p3, p2=p1, p3=p2 ] endmodule

// labels
label "hungry" = ((p1>0)&(p1<8))|((p2>0)&(p2<8))|((p3>0)&(p3<8));
label "eat" = ((p1>=8)&(p1<=9))|((p2>=8)&(p2<=9))|((p3>=8)&(p3<=9));
```

Ce morceau de code s'insère dans l'onglet *Model* du programme. Le nom du type de modèle y est simplement écrit en haut du programme. S'il s'était agit d'un dtmc, il aurait suffit d'écrire ce terme afin d'indiquer à Prism la nature du modèle à étudier.
Un module correspond à un processus qui — ici, est une instance de philosophe. Les mots-clés *module* et *endmodule* délimitent la zone de déclaration. Le philosophe 1 est représenté par la variable p1, qui pourra prendre 12 états différents, de 0 à 11, comme indiqué par la ligne *p1 : [0..11];*. Puis toutes les actions ainsi que probabilités sont simplement déclarées. Le fait qu'il y ait des actions multiples pour un état est modélisé par le fait d'écrire plusieurs ensembles de probabilités, comme pour ces deux lignes :
$[]p1 = 0 -> (p1' = 0)$;
$[]p1 = 0 -> (p1' = 1)$;

Deux modules supplémentaires sont créés ici, il y aura donc trois philosophes. Ceci modifie légèrement la procédure de calcul au niveau de Prism : puisqu'il y a plusieurs instances, Prism doit faire des produits de modèles avant de débuter ses calculs. Chaque état sera un vecteur d'états de toutes les instances en cours d'exécution.

Par ailleurs, deux labels sont déclarés, qui permettent de s'assurer que l'un des philosophes a faim, ou que l'un est en train de manger. Il est tout à fait possible d'en ajouter. Deux formules également permettent de tester si certaines actions sont possibles : *lfree* signifie que la fourchette gauche est libre, ce qui est possible seulement dans certains états de ses voisins. Il faut imaginer une table avec trois philosophes, et constater que les voisins ne sont évidemment pas les mêmes pour tous, ce pourquoi dans les deux modules supplémentaires créés, les variables ne sont pas les mêmes : *module phil2 = phil1 [p1=p2, p2=p3, p3=p1] endmodule*. Ceci permettra d'employer la même formule pour tous les philosophes.

Ce modèle, décrit dans le langage de Prism, est le même que celui décrit par la figure au début de ce document, l'algorithme y est le même : les philosophes attendent un temps indéterminé, et se mettent à essayer de manger. Un tirage au sort entre la gauche et la droite permet d'éviter le déterminisme, et empêche dans une grande partie des cas les interblocages. Dans un premier temps, il est possible de vérifier que les probabilités sont bien calculées. Pour ce faire, il faut aller dans la section *Properties* de Prism, et ajouter une nouvelle propriété.

Pour tester la probabilité minimum d'atteindre l'état « manger », il faut ajouter *Pmin= ? [F "eat"]*, puis demander *verify*. Ceci aura pour effet de lancer la procédure de calcul qui a été expliquée plus haut. Ici, le résultat est 0, comme attendu. La probabilité maximum s'obtient de la même façon, en entrant la ligne *Pmax= ? [F "eat"]* dans les *properties*, et la probabilité retournée est 1.

Cet exemple mobilisait trois instances de philosophes. Afin de comparer les temps de calcul, il suffit d'ajouter des philosophes, et de lancer les mêmes calculs. Par exemple, avec 30 instances de philosophes, le temps de modélisation

est bien plus important.
Pour simple comparaison, voici des temps d'exécutions sur une machine équipée d'un processeur *Intel i5-5200U* cadencé à 2.20GHz, avec 8 Go DDR3L de mémoire ram, sous le système d'exploitation *Ubuntu 16.04 Desktop*. Pour le modèle lancé avec 3 instances, la méthode de calcul est la méthode itérative, et le test *Pmin= ? [F "eat"]*, les logs de Prism indiquent :

Reachability (BFS) : 18 iterations in 0.00 seconds
Time for model construction : 0.021 seconds.
Prob0E : 1 iterations in 0.00 seconds
Prob1A : 1 iterations in 0.00 seconds
yes = 240, no = 716, maybe = 0
Time for model checking : 0.001 seconds.
Result : 0.0 (value in the initial state)
Avec 30 philosophes, et la même question, les logs indiquent cette fois-ci :

Reachability (BFS) : 180 iterations in 91.59 seconds
Time for model construction : 92.016 seconds.
Prob0E : 1 iterations in 0.16 seconds
Prob1A : 1 iterations in 0.08 seconds
yes = 6.147956459395484E29, no = 3.5116763806009458E28, maybe = 0
Time for model checking : 0.27 seconds.
Result : 0.0 (value in the initial state)
Avec 30 philosophes, la même question, mais la méthode de programmation linéaire, après plus d'une heure, l'application n'a pas réussi à obtenir de résultat, ce qui confirme l'information selon laquelle cette méthode n'est pas efficace pour les exemples exhaustifs.

Il ressort de ces résultats que l'étape la plus longue est celle de la construction du modèle, à côté de laquelle la vérification prend un temps significativement plus court.

La question posée permet de voir si l'état « eat » a un chance de se produire. Une autre question intéressante à se poser par rapport à ce modèle est de savoir s'il permet d'éviter les interblocages. Ceci peut être vérifié avec une autre *property* : $filter(forall,' hungry' => P >= 1[F'eat'])$.
Cette ligne signifie que pour chaque état *hungry*, la probabilité qu'un philosophe mange est au moins égale à 1. En d'autres termes, cela ne garantit pas que le philosophe qui a faim va manger, mais que des philosophes ne risquent pas de se retrouver en situation d'interblocage.
Appliqué au modèle avec trois instances, Prism indique que la propriété n'est pas vérifiée, qu'il existe des états qui ne la vérifient pas. Cela indique que la solution qui consiste à être « équitable », c'est à dire où seuls les philosophes qui ont faim essaient de manger ne garantit pas qu'il n'y aura pas d'interblocages. Ce résultat était connu, et il est confirmé ici par notre exemple.

Il existe une solution pour éviter totalement les interblocages. Il s'agit de ne plus appliquer la règle d'équité, et également, d'autoriser l'algorithme à ignorer un des philosophes. La différence se trouve dans la définition du module :

```
module phil1

    p1: [0..11];

//  [] p1=0 -> (p1'=0); // stay thinking
    [] p1=0 -> (p1'=1); // trying
    [] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly
    [] p1=2 &  lfree -> (p1'=4); // pick up left
//  [] p1=2 &  !lfree -> (p1'=2); // left not free
    [] p1=3 &  rfree -> (p1'=5); // pick up right
//  [] p1=3 &  !rfree -> (p1'=3); // right not free
    [] p1=4 &  rfree -> (p1'=8); // pick up right (got left)
    [] p1=4 & !rfree -> (p1'=6); // right not free (got left)
    [] p1=5 &  lfree -> (p1'=8); // pick up left (got right)
    [] p1=5 & !lfree -> (p1'=7); // left not free (got right)
    [] p1=6  -> (p1'=1); // put down left
    [] p1=7  -> (p1'=1); // put down right
    [] p1=8  -> (p1'=9); // move to eating (got forks)
    [] p1=9  -> (p1'=10); // finished eating and put down left
    [] p1=9  -> (p1'=11); // finished eating and put down right
    [] p1=10 -> (p1'=0); // put down right and return to think
    [] p1=11 -> (p1'=0); // put down left and return to think
endmodule
```

Dans ce cas, la vérification de la propriété $filter(forall,' hungry' => P >= 1[F'eat'])$ réussit, le modèle évite les interblocages.

Il est possible d'étudier beaucoup de choses encore dans ce modèle. Par exemple, Prism propose de simuler des exécutions, d'en extraire des graphiques. Il est aussi possible de demander au logiciel en combien de tours un philosophe qui a faim mange, ceci en ajoutant une constante dans les propriétés, et bien d'autres choses encore, qui sortent du cadre de ce travail.

Conclusion

Dans ce travail, plusieurs sujets ont été présentés : le model-checking, les MDPs, et Prism. Ceci a permis d'avoir un aperçu des possibilités d'un logiciel de vérification de modèle comme Prism, d'appréhender ses limites, ainsi qu'une partie de son implémentation. Le fait que le logiciel soit Open Source et bien documenté en fait un outil de référence, car la programmation est visible. Par ailleurs, ce travail met en évidence le problème du dîner des philosophes, ainsi que ses évolutions au cours du temps.
Le paysage actuel des logiciels de vérification de modèle se densifie, et cela est compréhensible principalement pour deux raisons : la première étant que les vérifications de modèle automatiques permettent d'avoir une bien meilleure connaissance de l'algorithme que l'on projette de développer, ainsi que de ses propriétés. Il est tout à fait imaginable qu'un modèle non valable soit ainsi rejeté avant des étapes de développement coûteuses. La seconde raison étant que l'automatisation, pour des tâches répétitives, comme pour les méthodes de calcul implémentées par Prism, est très utile et efficace : un ordinateur ne commet pas d'erreurs en répétant une tâche, même de nombreuses fois.

Acknowledgements

References

Berkeley (2014). Berkeley lazy abstraction software verification tool. http ://forge.ispras.ru/projects/blast/. dernier accès le 03/08/2016.

Dave Parker, Gethin Norman, M. K. (2016). Prism. http ://www.prismmodelchecker.org/. dernier accès le 03/08/2016.

Lehmann, D. and Rabin, M. (1981). On the advantage of free choice : A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138.

NASA (2016). Java path finder. http ://babelfish.arc.nasa.gov/trac/jpf/wiki. dernier accès le 03/08/2016.

Pagetti, C. (2014). Systèmes temps réel. http ://www.onera.fr/sites/default/files/u490/cours_temps_reel_in2-nup-nup.pdf. dernier accès le 02/08/2016.

Parker, D. D. (2011a). Model checking mdps. http ://www.prismmodelchecker.org/lectures/pmc/14-mdp

Parker, D. D. (2011b). Reachability in mdps. http ://www.prismmodelchecker.org/lectures/pmc/13-mdp

Prism (2010a). The prism language. http ://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction.

Prism (2010b). Tutoriel 6. http ://www.prismmodelchecker.org/tutorial/phil.php.

Quarteroni, A., Sacco, R., and Saleri, F. (2007). *Méthodes Numériques*. Springer, Paris, FR.

Sandler, K. (2010). Killed by code. http ://www.softwarefreedom.org/resources/2010/transparent-medical-devices.html. dernier accès le 03/08/2016.

Wikipedia (2016). Vol 501 d'ariane 5. https :fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5. dernier accès le 01/08/2016.

Y. De Smet, B. F. (2013). Algorithmique 3 et recherche opérationnelle. http ://homepages.ulb.ac.be/ bfortz/algo3.pdf.