Modélisation et vérification

Hamza Benkabbou, Kevin Bourgeois, Jérémy Morosi,
Jean-Baptiste Perrin, Zo Rabarijaona

25 janvier 2013

Table des matières

1 Organisation du code

1.1 principal

Ce package contient la classe `Main` du programme. Elle implémente l'interface `ICallback` (listing 1) qui définie toutes les fonctions demandées dans le sujet. L'intérêt de cette interface est de pouvoir passer le `Main` en paramètre du parseur `CommandLineParser`. Il a été créé avec ANTLR dans le but de pouvoir analyser simplement les commandes saisies par l'utilisateur et de pouvoir appeler les méthodes correspondantes dans le `Main` au travers de l'interface.

1.2 CTL

C'est le package qui contient la classe `CTL`. Outre les méthodes pour calculer la valeur d'une formule, on y trouve la méthode `valeur` qui retourne la liste des états validant une formule et la méthode `justifie` qui retourne la preuve.

1.3 rdp

C'est le package qui contient la classe `RdP`.

1.4 systeme

C'est le package qui contient les classes `AlgoGrapheRdP` et `GrapheRdP`. La classe `GrapheRdP` contient les méthodes `ctlToDot` pour afficher les états validant une formule au format `.dot` et `justifieToDot` pour la justification d'un état.

1.5 preuve

Ce package contient toutes les classes liées aux preuves. On y trouve les interfaces `IPreuve` (listing 2) et `IChemin` (listing 4) qui définissent ce qu'est une preuve. On y trouve aussi une implémentation pour chaque type de preuve et les deux classes `Coloration` (listing 3) et `Couleur` qui gèrent la coloration. La classe `Preuve` contient, entre autres, la méthode `neg` qui retourne la négation d'une formule.

2 Validation

Cette partie décrit comment la formule est analysée et validée.

2.1 Analyse

L'analyse de la formule saisie par l'utilisateur dans le terminal se fait à l'aide d'un parseur généré par ANTLR. Grâce à ce parseur, toute formule est mise sous la forme d'un arbre (classe `Tree`) où chaque branche est un morceau de la formule originale et les feuilles sont les formules atomiques qui la composent. Pour $EX(\$A \parallel \$B)$ par exemple, on obtiendrait le résultat de la figure 1. Il est ainsi plus facile d'utiliser la formule par la suite puisqu'il suffira de faire un `switch` sur le type d'un noeud et d'accéder à ses enfants en conséquence.

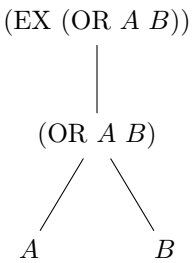


FIGURE 1 – Arbre de la formule $EX(\$A \parallel \$B)$

2.2 États valides

La validation a lieue dans la méthode `justifie` de la classe `CTL`. Comme dit précédemment, pour parcourir la formule on utilise un `switch` et on compare le type du noeud avec les constantes identifiants les tokens dans le parseur. Pour calculer la liste des états valides, il suffit donc de partir de la racine de l'arbre, atteindre les feuilles de manière récursive, récupérer les états valides pour les formules atomiques et appliquer chaque partie de la formule petit à petit en retournant vers la racine. À la fin on dispose de la liste de tous les états validant la formule.

3 Justification

Cette partie décrit la manière dont le programme calcule et mémorise la justification d'une formule CTL pour ensuite en donner une représentation visuelle.

3.1 Preuve

L'interface `IPreuve` (listing 2) définie les méthodes nécessaires pour prouver qu'une formule CTL est vraie.

La structure d'une preuve est très simple, puisqu'elle consiste en la formule qui lui est associée (`getFormule`), en la liste des états qui sont vrais (`getMarquage`, `setMarquage`) et en la liste des sous-preuves pour chaque morceau de la formule (`getPreuves`).

Une preuve dispose aussi de plusieurs méthodes pour la visualiser. On peut récupérer la formule qui lui est associée au format textuel grâce à la méthode `formuleToString`. On peut obtenir un affichage sous forme d'arbre de la preuve et de ses sous-preuves grâce à la méthode `toTree`. On peut récupérer la formule sous la forme d'un label coloré pour l'exportation au format `.dot` grâce à la méthode `toDotLabel`. Et finalement, on peut exporter la preuve au format `.dot` grâce aux méthodes `toDotRacine` et `toDot`.

Lors du calcul de la preuve, les méthodes `couperRacine` et `couper` permettent de supprimer les états qui sont inutiles car ils n'ont pas de prédécesseurs dans la preuve parent.

3.1.1 Création

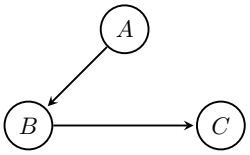
La création d'une preuve se fait en même temps que la validation de la formule CTL associée. Si la formule est un ensemble de sous-formules, on commence d'abord par les justifier. Ensuite, on peut créer une preuve du même type que la formule, en lui donnant le marquage la validant et la liste des sous-preuves. Au passage, on génère la couleur et le label (au format `.dot`) de la formule. Le label est l'ensemble des labels des sous-formules auxquels on ajoute le nom de la formule avec la couleur correspondante. Par exemple, pour $E(\$A \cup \$B)$, on obtient le label `<FONT COLOR="...">E(... U ...)</FONT>`.

3.1.2 Coupage

Une fois qu'on a construit la preuve complète, on appelle la méthode `couperRacine` en donnant en paramètre le numéro de l'état pour lequel on doit justifier la formule CTL. Cet appel a pour conséquence de mettre tous les états validant la preuve à $false$, sauf celui donné. La preuve appelle ensuite la méthode `couper` des sous-preuves en donnant son marquage en paramètre. Les sous-preuves vont alors remettre à $false$ tous les états valident qui n'ont pas de prédécesseurs dans le marquage donné, puis, elles vont à leur tour appeler la méthode pour leurs sous-preuves en donnant leur propre marquage. Ainsi, à la fin, il ne restera dans les preuves que les états qui sont réellement nécessaires à la justification.

3.1.3 Affichage au format textuel

L'affichage de la preuve au format textuel est assez simple. Une fois que tous les états non nécessaires ont été retirés par le coupage, on peut afficher la formule associée à la preuve, suivie de la liste des états la validant. Les sous-preuves sont affichées récursivement en appelant leur méthode `toTree` avec le niveau d'indentation désiré. On obtient alors un arbre dont la racine est la preuve complète et les dont les branches sont les formules atomiques (figure 2).



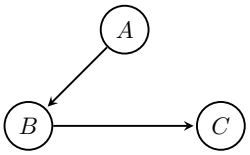
(a) Graphe d'états

```
EX(($B && EX($C))) = { 0 }
  ($B && EX($C)) = { 1 }
    $B = { 1 }
    EX($C) = { 1 }
      $C = { 2 }
```

(b) Preuve au format textuel

FIGURE 2 – Exemple de preuve pour la formule $EX(\$B \;\&\&\; EX(\$C))$ sur l'état 0

Certaines preuves plus complexes, comme les chemins, affichent les preuves pour tous les états du chemin avec le même niveau d'indentation et les entourent par des accolades (figure 3).



(a) Graphe d'états

```
E(($A || $B) U $C) = { 0 } {
  ($A || $B) = { 0 }
    $A = { 0 }
  ($A || $B) = { 1 }
    $B = { 1 }
  $C = { 2 }
}
```

(b) Preuve au format textuel

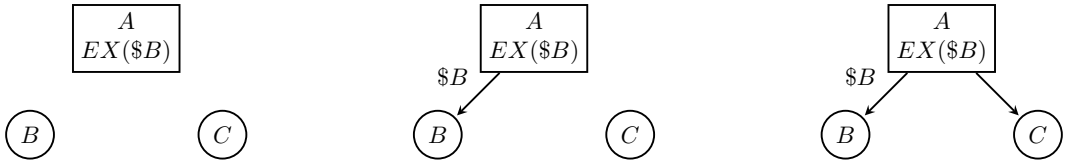
FIGURE 3 – Exemple de preuve pour la formule $E((\$A \parallel \$B) \cup \$C)$ sur l'état 0

3.1.4 Affichage au format .dot

L'exportation de la preuve au format `.dot` commence dans la méthode `justifieToDot` de la classe `GrapheRdP`. On appelle la méthode `toDotRacine` de la preuve complète en lui donnant en paramètres une variable de type `Map<Integer, Set<Integer>>` et une variable de type `Set<String>`. La première, dont les clefs sont les états du réseau de pétri et les valeurs sont les listes des états auxquels ils sont reliés, permettra à toutes les sous-preuves d'indiquer quels états sont reliés par une justification. La seconde devra contenir l'ensemble des flèches du graphe (au format `.dot`) qui font parties de la justification.

La méthode `toDotRacine` sert uniquement à appeler la méthode `toDot` pour toutes les sous-preuves sans qu'une flèche ne soit ajoutée pour l'état de départ. La méthode `toDot` permet à une preuve d'ajouter toutes les flèches nécessaires au graphe puis de s'appeler pour toutes les sous-preuves. Une flèche doit partir d'un état parent, aller vers un état validant la sous-preuve et elle doit avoir le label de la formule associée à la sous-preuve à côté.

Une fois que toutes les flèches ont été ajoutées, on commence par exporter la liste des états avec leur marquage et avec le label de la formule complète pour l'état pour lequel on doit justifier la formule (figure 4a). On exporte ensuite l'ensemble des flèches qui se trouvent dans la variable précédente (figure 4b). Puis on complète le graphe avec les flèches manquantes (celles qui ne font pas parties de la justification) (figure 4c).



(a) Preuve complète (b) Sous-preuves (c) Flèches manquantes

FIGURE 4 – Affichage de la preuve pour $EX(\$B)$

Pour l'ajout des flèches manquantes, on utilise la liste des états reliés pour savoir si elle existe déjà. La liste des flèches nous permet de nous assurer qu'une même preuve ne soit affichée qu'une fois (figure 5).

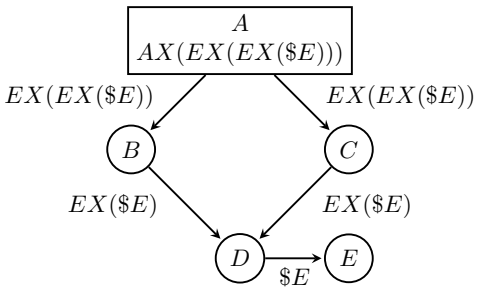


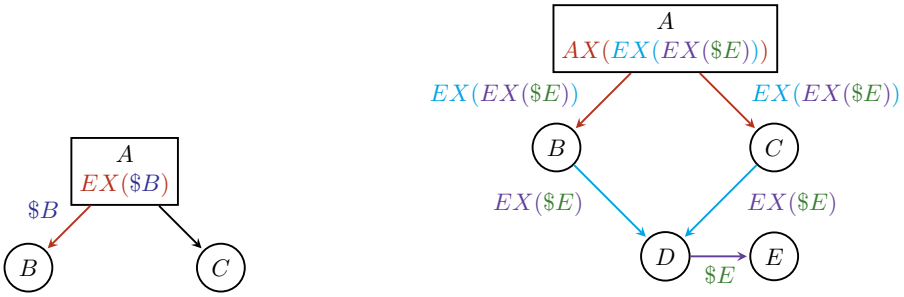
FIGURE 5 – Affichage de la preuve pour $AX(EX(EX(\$E)))$

3.1.5 Coloration

La coloration des preuves est gérée par la classe `Coloration` (listing 3). Son but est d'associer une couleur unique (si possible) et un label coloré (pour l'exportation au format `.dot`) à une formule. Lors de l'exportation, les accesseurs `getCouleur` et `getLabel` permettront alors de récupérer les informations liées à la formule donnée.

Les couleurs sont générées par la méthode `genererCouleur`. Dans l'idéal, toutes les couleurs utilisées doivent être uniques et ne doivent pas trop se ressembler pour que la preuve soit suffisamment claire. Ici, on génère juste des couleurs au format HSV avec $s = 1$, $v = 0.75$ et en incrémentant le h d'une certaine valeur à chaque appel de la méthode.

La couleur d'une formule est utilisée pour colorer une flèche partant d'un état validant la formule et allant vers un état validant une sous-formule alors que le label de la sous-formule est affiché sur la flèche (figure 6).



(a) $EX(\$B)$ (b) $AX(EX(EX(\$E)))$

FIGURE 6 – Exemples de colorations

3.2 Preuves atomiques

Les preuves atomiques sont les preuves associées aux formules atomiques p, $true$, $false$, $dead$ et $initial$. Elles correspondent respectivement aux classes `Atom`, `True`, `False`, `Dead` et `Initial`. Comme leur implémentation est assez simple puisqu'elle n'ont pas de sous-preuves, on montrera juste les affichages obtenus au format textuel (figure 7) et au format `.dot` (figure 8) (sauf pour $false$ qui ne peut pas être affichée).

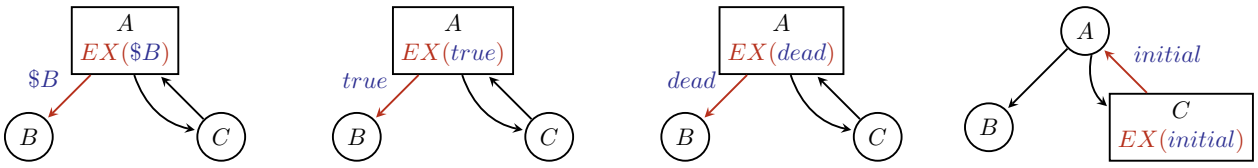
```
EX($B) = { 0 }
  $B = { 1 }
```
(a) $EX(\$B)$

```
EX(true) = { 0 }
  true = { 1 }
```
(b) $EX(true)$

```
EX(dead) = { 0 }
  dead = { 1 }
```
(c) $EX(dead)$

```
EX(initial) = { 2 }
  initial = { 0 }
```
(d) $EX(initial)$

FIGURE 7 – Affichages au format textuel obtenus pour les preuves atomiques



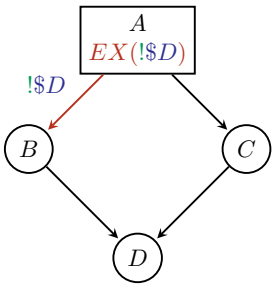
(a) $EX(\$B)$ (b) $EX(true)$ (c) $EX(dead)$ (d) $EX(initial)$

FIGURE 8 – Affichages au format `.dot` obtenus pour les preuves atomiques

3.3 Preuves successeurs

3.3.1 *EX*

Lors du coupage, une preuve de type EX ne va garder qu'un seul état valide de la sous-preuve puisqu'il suffit de prouver que la formule est vraie pour un état voisin (figure 9).



(a) Affichage `.dot`

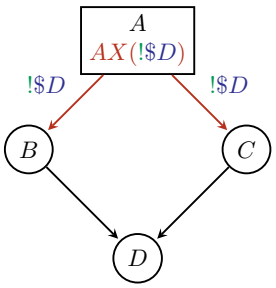
```
EX(!$D) = { 0 }
  !$D = { 1 }
    $D = { 2 }
```

(b) Affichage textuel

FIGURE 9 – Preuve de $EX(!\$D)$ pour l'état 0

3.3.2 *AX*

Lors du coupage, une preuve de type AX va dupliquer plusieurs fois la sous-preuve pour chaque état la validant et étant un successeur (figure 10). On simplifie ainsi les choses car une sous-preuve qui avait plusieurs états valides se retrouve en plusieurs exemplaires avec un seul état à chaque fois. Les autres preuves plus complexes seront donc justifiées uniquement sur un état.



(a) Affichage `.dot`

```
AX(!$D) = { 0 }
  !$D = { 1 }
    $D = { 2 }
  !$D = { 3 }
    $D = { 2 }
```

(b) Affichage textuel

FIGURE 10 – Preuve de $AX(!\$D)$ pour l'état 0

3.4 Preuves chemins

Les preuves qui consistent en un chemin d'états implémentent l'interface `IChemin` (listing 4) qui est elle-même basée sur `IPreuve`.

La structure d'un chemin consiste en la preuve pour l'état de départ (`getDebut`), celle pour l'état d'arrivée (`getFin`) et la liste des états par lesquels on passe (`getChemins`). Il n'y a qu'un seul état d'arrivée car, par exemple, si on doit prouver $AF(\cdots)$ pour plusieurs états, on va décomposer la preuve en plusieurs sous-preuves $AF(\cdots)$ pour chaque état (figure 15).

```
AF($B2) = { 1 4 } {
  AF($B2) = { 1 } {
    ...
  }
  AF($B2) = { 4 } {
    ...
  }
}
```

FIGURE 11 – Décomposition de la preuve de $AF(\$B2)$ pour chaque état

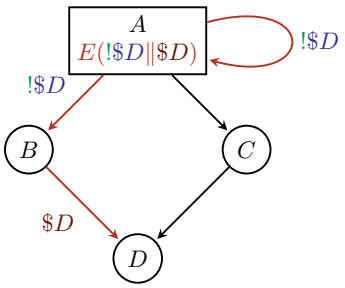
Un chemin dispose aussi des accesseurs `estFin` et `aVoisinFin` qui lui permettent de savoir si il doit continuer la preuve pour les états voisins ou non.

3.4.1 *EU*

Après avoir été créée, la preuve de type *EU* contient la preuve pour les états validant la condition du chemin, celle pour les états validant la condition d'arrêt et la liste des états qui ont un tel chemin.

Le coupage de la preuve entraîne la création d'une liste de sous-preuves pour tous les états par lesquels le chemin passe. Pour ce faire, on sait que la preuve parent a été coupée et qu'on doit prouver le chemin à partir des états restants. On va donc utiliser la liste des états ayant un tel chemin et ne garder que ceux qui ont pour prédécesseurs un ou plusieurs états de départ. On continue d'appliquer cette technique jusqu'à avoir atteint un des états validant la condition d'arrêt du chemin. Une fois que c'est fait, on peut supprimer tous les états du chemin qui ne sont pas nécessaires.

L'affichage d'une preuve de type *EU* consiste donc en la preuve de la condition de départ pour chaque état du chemin et de la condition d'arrêt pour l'état final (figure 12). Pour l'affichage au format textuel, la liste des états est affichée entre deux accolades et chaque étape du chemin possède la même indentation (figure 12b).



(a) Affichage `.dot`

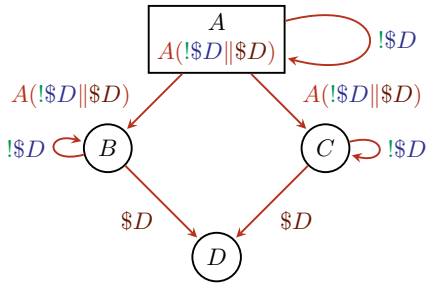
```
E(!$D U $D) = { 0 } {
  !$D = { 0 }
    $D = { 2 }
  !$D = { 1 }
    $D = { 2 }
  $D = { 2 }
}
```

(b) Affichage textuel

FIGURE 12 – Preuve de $E(!\$D \cup \$D)$ pour l'état 0

3.4.2 *AU*

Tout comme pour la preuve de type *EU*, la preuve de type *AU* va créer une liste de sous-preuves pour chaque états du chemin. Cependant, cette liste consistera en la dupplication de la preuve pour chacun de ces états. Ainsi, pour chaque état on prouvera que pour tous les états voisins il existe aussi le chemin (figure 13).



(a) Affichage `.dot`

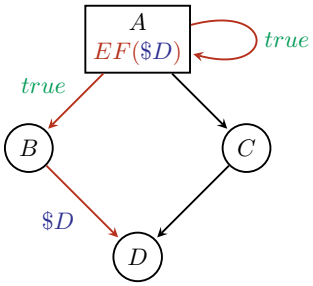
```
A(!$D U $D) = { 0 } {
  !$D = { 0 }
    $D = { 2 }
  A(!$D U $D) = { 1 } {
    !$D = { 1 }
      $D = { 2 }
    $D = { 2 }
  }
  A(!$D U $D) = { 3 } {
    !$D = { 3 }
      $D = { 2 }
    $D = { 2 }
  }
}
```

(b) Affichage textuel

FIGURE 13 – Preuve de $A(!\$D \cup \$D)$ pour l'état 0

3.4.3 *EF*

Une preuve de type *EF* est une preuve similaire à celle de type *EU* à la différence qu'elle n'a pas de condition à vérifier sur l'ensemble du chemin. Ainsi, on utilise une preuve *EU* pour laquelle tous les états du chemin doivent vérifier *true* (figure 14).



(a) Affichage `.dot`

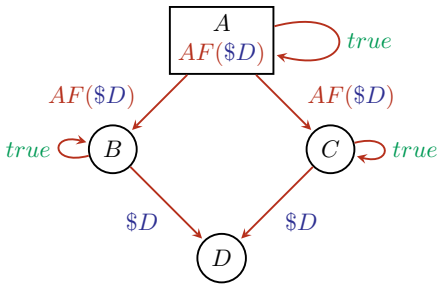
```
EF($D) = { 0 } {
  true = { 0 }
  true = { 1 }
  $D = { 2 }
}
```

(b) Affichage textuel

FIGURE 14 – Preuve de $EF(\$D)$ pour l'état 0

3.4.4 *AF*

Comme pour la preuve de type *EF*, la preuve de type *AF* utilise une preuve *AU* pour laquelle tous les états du chemin doivent vérifier *true* (figure 15).



(a) Affichage `.dot`

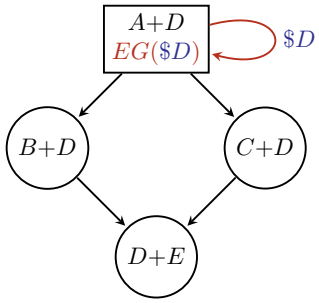
```
AF($D) = { 0 } {
  true = { 0 }
  AF($D) = { 1 } {
    true = { 1 }
    $D = { 2 }
  }
  AF($D) = { 3 } {
    true = { 3 }
    $D = { 2 }
  }
}
```

(b) Affichage textuel

FIGURE 15 – Preuve de $AF(\$D)$ pour l'état 0

3.4.5 *EG*

La preuve de type *EG* utilise une preuve *EU* pour laquelle tous les états du chemin doivent vérifier la condition (figure 16).



(a) Affichage `.dot`

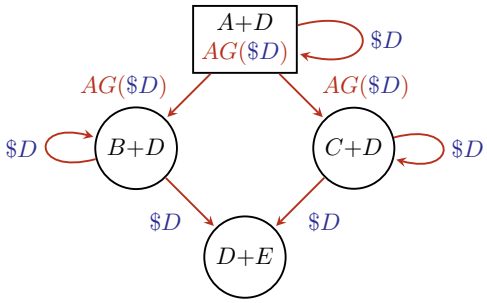
```
EG($D) = { 0 } {
  $D = { 0 }
}
```

(b) Affichage textuel

FIGURE 16 – Preuve de $EG(\$D)$ pour l'état 0

3.4.6 *AG*

La preuve de type *AG* utilise une preuve *AU* pour laquelle tous les états du chemin doivent vérifier la condition (figure 17).



(a) Affichage `.dot`

```
AG($D) = { 0 } {
  $D = { 0 }
  AG($D) = { 1 } {
    $D = { 1 }
    $D = { 2 }
  }
  AG($D) = { 3 } {
    $D = { 3 }
    $D = { 2 }
  }
}
```

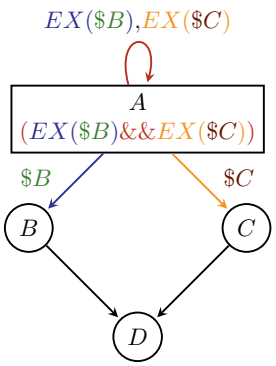
(b) Affichage textuel

FIGURE 17 – Preuve de $AG(\$D)$ pour l'état 0

3.5 Preuves opérateurs

3.5.1 &&

La preuve de type && affiche les preuves des deux conditions pour les états qui la valident (figure 18).



(a) Affichage `.dot`

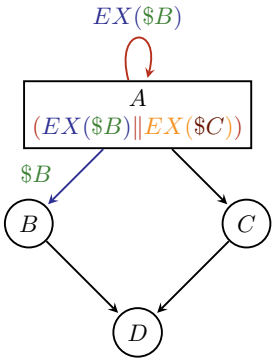
```
(EX($B) && EX($C)) = { 0 }
  EX($B) = { 0 }
    $B = { 1 }
  EX($C) = { 0 }
    $C = { 3 }
```

(b) Affichage textuel

FIGURE 18 – Preuve de $EX(\$B)$ && $EX(\$C)$ pour l'état 0

3.5.2 ||

La preuve de type || affiche la preuve d'une des deux conditions dans le cas où elles seraient toutes les deux valides ou bien la preuve de celle qui est vraie (figure 19).



(a) Affichage `.dot`

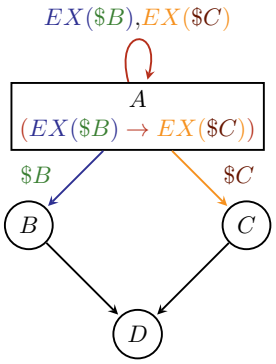
```
(EX($B) || EX($C)) = { 0 }
  EX($B) = { 0 }
    $B = { 1 }
```

(b) Affichage textuel

FIGURE 19 – Preuve de $EX(\$B) \parallel EX(\$C)$ pour l'état 0

3.5.3 →

L'implication est affichée comme le && (figure 20).



(a) Affichage `.dot`

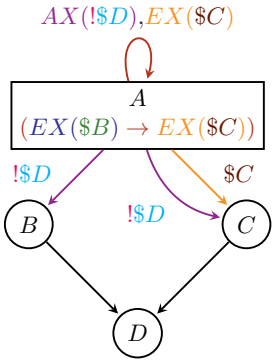
```
(EX($B) -> EX($C)) = { 0 }
  EX($B) = { 0 }
    $B = { 1 }
  EX($C) = { 0 }
    $C = { 3 }
```

(b) Affichage textuel

FIGURE 20 – Preuve de $EX(\$B) \rightarrow EX(\$C)$ pour l'état 0

Cependant, avec l'implication, il peut n'y avoir aucun état validant la condition de gauche et dans ce cas on

devra quand même prouver que l'implication est correcte. Pour ce faire, si la condition de gauche n'a aucun état valide alors on calcule la preuve de sa négation, et si la condition de droite est également fausse alors on calcule aussi la preuve de sa négation. Ainsi, l'affichage sera la preuve que la ou les conditions ne sont pas valides (figure 21).



(a) Affichage `.dot`

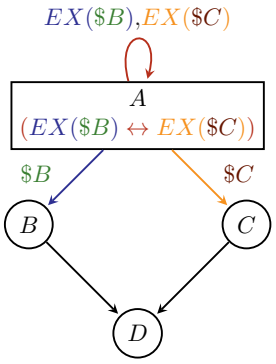
```
(EX($D) -> EX($C)) = { 0 }
  AX(!$D) = { 0 }
    !$D = { 1 3 }
      $D = { 2 }
  EX($C) = { 0 }
    $C = { 3 }
```

(b) Affichage textuel

FIGURE 21 – Preuve de $EX(\$D) \rightarrow EX(\$C)$ pour l'état 0

3.5.4 ↔

L'équivalence est affichée comme le && (figure 22). Et comme pour l'implication, si aucun état n'est valide pour les deux conditions alors on calcule les preuves de leur négation et on les affiche à la place.



(a) Affichage `.dot`

```
(EX($B) <-> EX($C)) = { 0 }
  EX($B) = { 0 }
    $B = { 1 }
  EX($C) = { 0 }
    $C = { 3 }
```

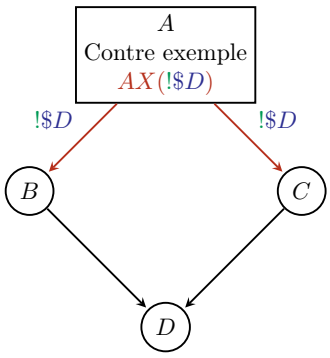
(b) Affichage textuel

FIGURE 22 – Preuve de $EX(\$B) \leftrightarrow EX(\$C)$ pour l'état 0

3.6 Contre exemple

Lorsque l'état indiqué par l'utilisateur n'est pas un état valide de la formule saisie, on affichage à la place la justification d'un contre exemple en partant du même état. Pour trouver ce contre exemple, on fait simplement la négation de la formule en parcourant de manière récursive l'arbre retourné par le parseur. On dispose alors d'une nouvelle formule au format textuel que l'on va pouvoir utiliser, comme si l'utilisateur l'avait entrée, pour afficher la justification.

Dans l'exemple ci-dessous (figure 23b), la formule $EX(\$D)$ n'était pas valide pour l'état 0 et a donc été transformée en la formule $AX(!\$D)$ qui est la négation de la première. Dans la console, l'utilisateur est prévenu par un message lui disant que la formule donnée n'était pas valide. Dans le graphe (figure 23a), le message `Contre exemple` est affiché sous le label de l'état de départ.



(a) Affichage `.dot`

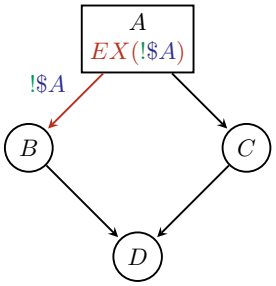
```
> Justifie EX($D) 0
L'état donné ne valide pas la formule.
Justification de AX(!$D) pour l'état 0.
AX(!$D) = { 0 }
  !$D = { 1 }
    $D = { 2 }
  !$D = { 3 }
    $D = { 2 }
```

(b) Affichage textuel

FIGURE 23 – Contre exemple de $EX(\$D)$ pour l'état 0

3.7 Preuve négation

La preuve de la négation est similaire aux contre exemples. Si la négation est appliquée à une formule atomique et que la visualisation est donc directe, on se contente d'afficher la preuve telle quelle (figure 24). Par contre, si la négation est appliquée à une formule complexe, on commence par récupérer la négation de la formule, puis on la justifie comme si elle avait été saisie par l'utilisateur (figure 25). Lors de la coupure, on dira que l'état de départ de cette nouvelle formule est l'état validant la négation et ainsi on obtiendra la justification de la négation pour l'état voulu.

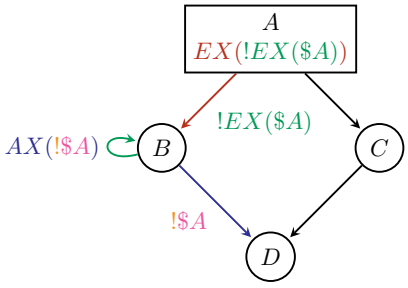


(a) Affichage `.dot`

```
EX(!$A) = { 0 }
  !$A = { 1 }
    $A = { }
```

(b) Affichage textuel

FIGURE 24 – Preuve de $EX(\$A)$ pour l'état 0



(a) Affichage `.dot`

```
EX(!EX($A)) = { 0 }
  !EX($A) = { 1 }
    AX(!$A) = { 1 }
      !$A = { 2 }
        $A = { }
```

(b) Affichage textuel

FIGURE 25 – Preuve de $EX(!EX(\$A))$ pour l'état 0

4 Utilisation du programme

4.1 Invité de commande

Par défaut, le programme s'utilise en invité de commande. Dans ce mode, il affiche le symbole `>` et attend la saisie d'une commande. Les commandes données sont parsées à l'aide d'un parseur généré par ANTLR (voir

`principal/CommandLine.g`) qui va se charger d'appeler les méthodes correspondantes dans la classe `Main`.

Les commandes disponibles sont toutes celles demandées dans le sujet ainsi que quelques unes qui ont été ajoutées.

4.2 Commandes

load	<fichier.net>	charge le réseau de pétri depuis le fichier
graphe		calcule le graphe des états accessibles
look	<etat>	affiche le marquage de l'état
succ	<etat>	affiche la liste des successeurs de l'état
todot	<fichier.dot>	exporte le graphe au format `.dot` dans le fichier
ctl	<formule>	affiche le nombre d'états validant la formule
ctl	<formule> <etat>	affiche *vrai* si l'état valide la formule ou $faux$
ctltodot	<formule> <fichier.dot>	exporte le graphe au format `.dot` en colorant les états qui valident la formule
Justifie	<formule> <etat>	affiche la preuve au format textuel que l'état valide la formule
Justifietodot	<formule> <etat> <fichier.dot>	exporte le graphe et la preuve que l'état valide la formule au format `.dot` dans le fichier
shell		passe en mode shell
stop		arrête le programme

4.3 Création d'un script

Outre le mode invité de commande, il est possible de créer un script pour lancer le programme et lui faire exécuter un ensemble de commandes. Les différents exemples fournis avec le projet fonctionnent ainsi :

On lance le programme normalement avec `java -jar modelprojet.jar` et on lui donne une liste de paramètres entre les deux balises `END_PARAMS`.

```
java -jar modelprojet.jar << -END_PARAMS
    ...
END_PARAMS
```

On commence par exécuter la commande `shell` qui permet au programme de passer en mode shell. Dans ce mode, toutes les commandes exécutées sont affichées après le symbole `>`. Ce mode est nécessaire car les commandes fournies au programme entre les balises `END_PARAMS` ne sont pas affichées dans la console comme si elles avaient été saisies par l'utilisateur. La dernière commande à exécuter doit être la commande `stop` pour demander au programme de s'arrêter une fois le script fini.

```
java -jar modelprojet.jar << -END_PARAMS
    shell
    ...
    stop
END_PARAMS
```

C'est entre les deux commandes `shell` et `stop` que l'on peut ajouter toutes les commandes du script.

```
java -jar modelprojet.jar << -END_PARAMS
    shell
    load "hello.net"
    graphe; todot "hello.dot"
    ctl EX(EX(\$C)); ctl EX(\$C)
    stop
END_PARAMS
```

À noter que des commandes séparées par un retour à la ligne produiront le même effet que si l'utilisateur avait saisi la première, appuyé sur entrée puis avait saisi la seconde. À l'inverse, deux commandes séparées par une point-virgule seront parsées et exécutées à la suite comme si elles avaient été saisies en même temps par l'utilisateur.

La différence est que, dans le second cas, les résultats des deux commandes seront affichés à la suite sans savoir quelle commande a produit quel résultat. Alors que dans le premier cas, le résultat de la première commande sera séparé du résultat de la seconde par le symbole `>` et l'affichage de la seconde commande.

5 Exemples

De nombreux exemples ont été créés et sont disponibles dans le dossier d'exemples du projet. Ces exemples sont sous la forme d'un script `.sh` qui doit se trouver dans le même dossier que le `.jar` du programme.

Chaque exemple contient des commentaires indiquant son intérêt et les fichiers produits. Cependant, voici une liste récapitulative des plus importants :

Script `.sh`	Description
test commandes	exécute toutes les commandes demandées dans le sujet sur le fichier `hello.net`
test justifie	produit une justification au format textuel et `.dot` de l'ensemble des formules possibles sur les différents fichiers `.net`
preuves atomiques	produit une justification au format textuel et `.dot` des formules atomiques p, $true$, $dead$ et $initial$ sur le fichier `atomique.net`
preuves voisins	produit une justification au format textuel et `.dot` des formules atomiques EX et AX sur le fichier `chemin.net`
preuves chemins	produit une justification au format textuel et `.dot` des formules atomiques EU, AU, EF, AF, EG et AG sur les fichiers `chemin.net` et `chemin2.net`
preuves operateurs	produit une justification au format textuel et `.dot` des formules atomiques $\&\&, \|\|, \rightarrow$ et $\leftrightarrow$ sur le fichier `operateur.net`
contre exemple	évalue chaque formule $p, true, initial, \cdots, EX, \cdots, EF, \cdots$ sur un état du rdp `chemin.net` pour lequel elle n'est pas valide afin d'afficher le contre exemple.

Quelques exemples concrets :

Script `.sh`	Description
indiana	affiche, dans `indiana ctl.dot`, les états qui ont un chemin vers la solution et, dans `indiana justifie.dot`, la preuve qu'il est possible d'atteindre la solution depuis l'état 0
Chameaux3	même chose mais avec l'exemple des chameaux
aspirateur	affiche, dans `aspirateur ctl.dot`, les états qui ont un chemin vers un état où l'aspirateur a nettoyé les deux cases et, dans `aspirateur justifie.dot`, la preuve qu'il est possible d'atteindre un tel état depuis l'état 6

```
public interface ICallback {

  public RdP getRdP();
  public void shell();
  public void load(String filename);
  public void graphe();
  public void look(int etat);
  public void succ(int etat);
  public void toDot(String filename);
  public void ctl(Tree formule);
  public void ctl(Tree formule, int etat);
  public void ctlToDot(Tree formule, String filename);
  public void justifie(Tree formule, int etat);
  public void justifieToDot(Tree formule, int etat, String filename);
  public void stop();

}
```

Listing 1 – Interface `ICallback`

```
public interface IPreuve {

  public Tree getFormule();
  public boolean[] getMarquage();
  public void setMarquage(boolean[] marquage);
  public List<IPreuve> getPreuves();

  public void couperRacine(CTL ctl, int[][] pred, int etat);
  public void couper(CTL ctl, int[][] pred, boolean[] parents);

  public String toTree();
  public String toTree(String indent);

  public void toDotRacine(Map<Integer, Set<Integer>> fleches,
      Set<String> justifications, IPreuve parent, int etatParent,
      Coloration couleurs);
  public void toDot(Map<Integer, Set<Integer>> fleches,
      Set<String> justifications, IPreuve parent, int etatParent,
      Coloration couleurs);

  public String toDotLabel(Coloration couleurs);
  public IPreuve clone();
  public String formuleToString();

}
```

Listing 2 – Interface `IPreuve` commune à toutes les preuves

```
public class Coloration {

  private Map<Tree, String> couleursFormules;
  private Map<Tree, String> labelsFormules;

  public String getCouleur(Tree formule);
  public String getLabel(Tree formule);

  public void ajouter(Tree formule, String couleur, String label);
  public FakeTree ajouter(String label);

  public String genererCouleur();

}
```

Listing 3 – Classe `Coloration`

```java
public interface IChemin extends IPreuve {

  List<boolean[]> getChemins();

  IPreuve getDebut();
  void setDebut(IPreuve preuve);

  IPreuve getFin();
  void setFin(IPreuve preuve);

  boolean estFin();
  boolean aVoisinFin();

}
```

Listing 4 – Interface `IChemin`