Petit précis sur les outils Tina

La boîte à outils Tina [1] (TIme petri Net Analyzer) est un ensemble d'outils développés par le LAAS et qui permettent de manipuler des réseaux de Pétri temporisés.

Ce document a pour but de présenter quelques outils de Tina sur les réseaux de Pétri (non temporisés), afin de les utiliser dans le contexte de l'UE d'Ingénierie Dirigée par les Modèles.

Ce n'est pas un cours sur les réseaux de Pétri, ni sur le model-checking, ni sur la LTL, mais plut un genre de manuel technique sur l'utilisation des outils et ce que l'on peut leur faire dire.

1 Installation

Pour installer Tina le mieux est de passer par la page Download du site [2] et de sélectionner la version correspondant à votre OS.

Si vous êtes sur une distribution Linux, la commande suivante permet de télécharger la bonne archive directement :

```
> wget https://projects.laas.fr/tina/binaries/tina-3.7.0-amd64-linux.tgz
```

Une fois téléchargée, décompresser l'archive (`tar zxf tina-3.7.0-xxx.tgz`). Les outils de Tina se trouvent dans le répertoire `bin` (vous pouvez l'ajouter à votre `PATH` pour plus de simplicité).

2 L'outil `nd`

La documentation [3] présente en détail les outils Tina. C'est un écosystème assez riche et complexe, donc nous ne rentrerons pas dans les détails.

L'outil que nous utiliserons le plus est `nd` (Net Draw), qui permet notamment de dessiner des réseaux de Pétri, mais aussi d'appeler les autres outils avec une interface graphique.

2.1 Format d'entrée

Les outils Tina manipulent plusieurs formats de fichier. Celui qui va nous intéresser est le format `.net`, qui permet de définir un réseau de Pétri. Ce format présente 3 éléments :

— Le nom du réseau, introduit par le mot-clef `net` ; ce nom ne doit pas contenir d'espace

```
net mon_reseau
```

— Les places du réseau, données avec le mot-clef `pl` ; les places ont un nom et un *marquage* (nombre de jetons initiaux) optionnel, 0 étant le marquage par défaut

```
pl A_avec_0_jeton
pl B_avec_1_jeton (1)
```

1. https://projects.laas.fr/tina/index.php
2. https://projects.laas.fr/tina/download.php
3. https://projects.laas.fr/tina/papers.php

— Les transitions du réseau, spécifiées avec le mot-clef `tr` ; les transitions ont un nom, un intervalle de temps optionnel (`[x,y]` ou `[x,w[` si non borné), et se présentent sous la forme d'une flèche, avec à gauche la liste des places en entrée (séparées par un espace) et à droite la liste des places en sortie. Pour chaque place, on peut rajouter un poids optionnel avec `*x` pour donner le poids de l'arc associé (poids de 1 par défaut)

```
tr AB_vers_CD A*1 B*2 -> C*3 D*4
tr A_vers_B_tempo [3,5] A*6 -> B
tr C_vers_D_non_borne [4,w[ C -> D*2
```

Tina supporte les *read arcs* en remplaçant l'étoile par un point d'interrogation :

```
tr A_vers_B A?2 -> B*4
```

À noter qu'on peut laisser la liste de places, à gauche ou à droite de la flèche, vide :

```
tr vers_A -> A*2
tr B_vers B*3 ->
```

Dans la suite, considérons le fichier suivant :

```
net machin
pl A (7)
pl B (3)
pl C
pl D
tr t1 A*3 B*2 -> C
tr t2 A B -> D
tr t3 C D -> A*4 B*3
```

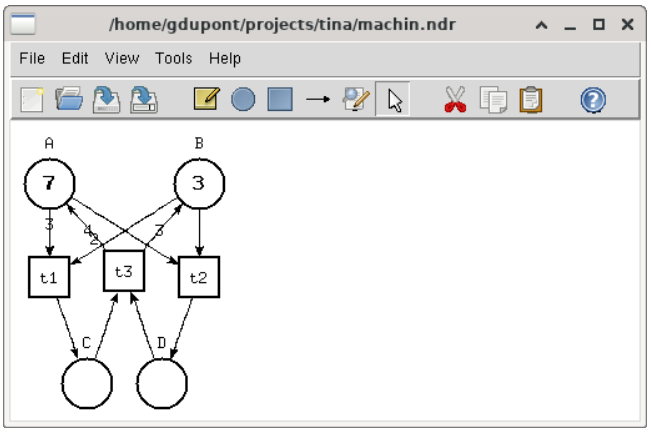
2.2 Lancer l'outil `nd`

On peut lancer `nd` directement sur un fichier `.net` :

```
> nd machin.net
```

L'outil nous notifie s'il y a des erreurs de syntaxe. S'il n'y a pas d'erreur, il se contente de montrer le contenu du fichier. À partir de là, on peut lancer plusieurs outils.

2.3 Dessiner le réseau

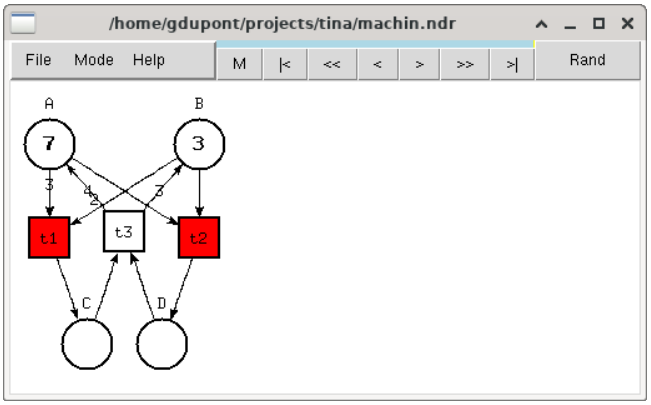


On peut afficher le réseau de Pétri sous forme de diagramme en faisant *Edit* > *draw*. L'outil

demande quelle technologie utiliser pour l'affichage ; elles sont globalement relativement équivalentes, donc vous pouvez laisser `neato`.

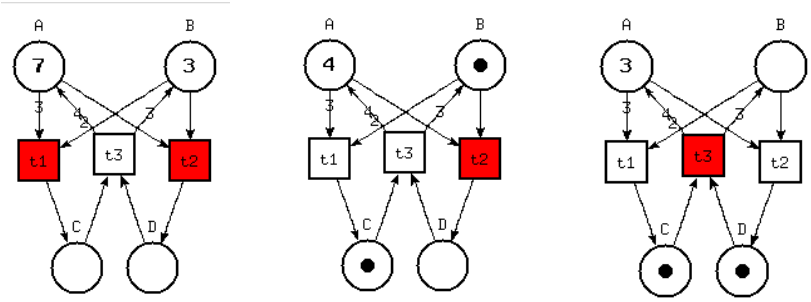
Une fois qu'on valide, on arrive sur l'éditeur de diagramme. À noter qu'on peut déplacer les éléments (pour rendre le diagramme plus lisible typiquement). Le menu `View` contient notamment pas mal de commandes pour améliorer l'affichage, à vous d'expérimenter !

2.4 Simulateur pas-à-pas



Il est possible de "simuler" le réseau à l'aide de *Tools* > *stepper simulator*.

L'outil montre alors le marquage actuel du réseau. Les transitions franchissables sont affichées en rouge et on peut cliquer dessus, ce qui déplace les jetons de place en place.



Le stepper est un moyen simple de regarder comment le réseau évolue. On peut même demander à Tina de choisir lui-même des transitions aléatoirement (bouton *Rand*), ce qui permet de voir rapidement si certains états empêchent de progresser plus loin (deadlock).

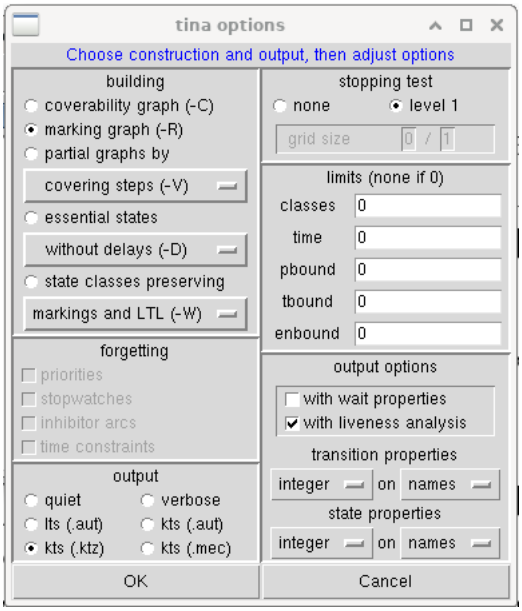
2.5 Analyse de l'espace d'état, graphe de marquage

On peut obtenir diverses informations sur les propriétés du réseau en demandant à `nd` une *analyse de l'espace d'état* (*Tools* > *state space analysis*).

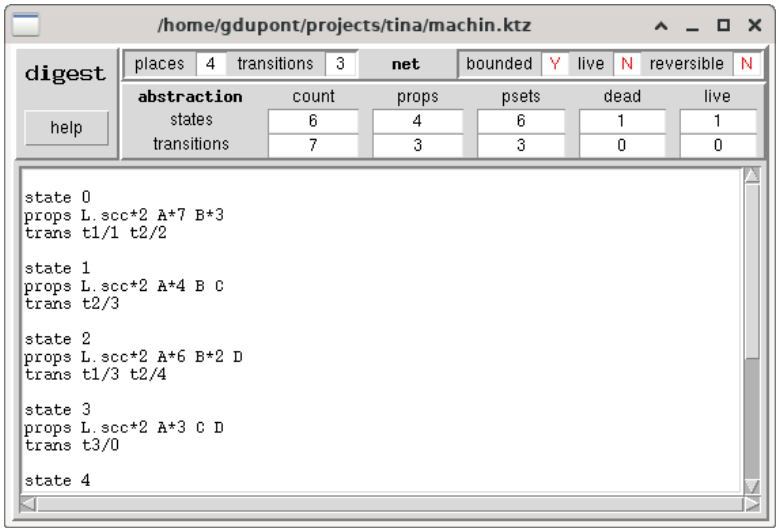
Dans la fenêtre de configuration, on peut choisir entre un graphe de marquage (*marking graph*) ou un graphe de couverture (*coverability graph*). La différence entre les deux est un peu subtile ; pour faire simple, si le réseau n'est pas borné, seul le graphe de couverture fonctionne. Sinon, le graphe de marquage donne généralement des résultats plus précis avec `selt`.

Sur cette même interface, on peut aussi cocher la case *with liveness analysis* pour demander à l'outil si notre réseau est vivace (= sans deadlock).

Les autres options de la fenêtre ne nous sont pas vraiment utiles ici, on laisse donc les valeurs par défaut.



Après validation, on obtient une fenêtre qui résume les données du réseau : nombre de places et de transitions, est-ce que le réseau est borné (*bounded*), vivace (*live*) et réversible (*reversible* = peut toujours revenir à l'état initial). Un "Y" dans la case signifie que la propriété est vraie (= *yes*), un "N" qu'elle est fausse (= *no*) et un "?" que l'outil n'a pas réussi à calculer la réponse.

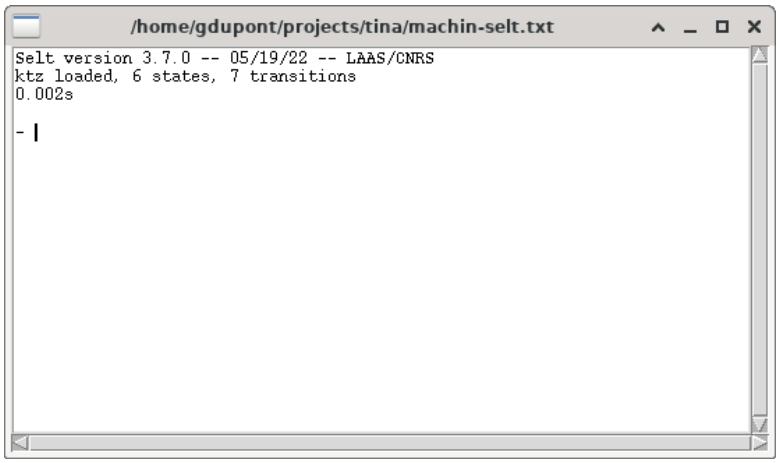


Dans l'en-tête de la fenêtre (zone *abstraction*), on a les propriétés du graphe de marquage (on rappelle qu'il s'agit, en fait, d'un automate) : nombre d'états, nombre de transitions. On a aussi notamment le nombre d'états pièges (*dead*), ou vivants (*live*).

En-dessous, on a le graphe de marquage sous forme textuelle.

3 Propriétés LTL

On peut cliquer dans la fenêtre du graphe de marquage et faire *model check LTL* pour faire apparaître une sorte de console, qui nous permet d'écrire des propriétés LTL à vérifier sur le réseau. L'outil qui s'affiche s'appelle `selt`.



Une formule `selt` se compose d'*opérateur* et de *symboles*, et se termine par un point-virgule (`;`). Une fois rentrée dans la console, on peut faire *Entrée* et l'outil nous répond, soit avec `TRUE` si la formule est vraie, soit avec `FALSE` et un contre-exemple si la formule est fausse. L'outil nous donne aussi le temps qu'il a pris à calculer la réponse, pour information.

À noter que tout ce qui suit le `;` n'est pas pris en compte par `selt` (si vous voulez mettre des commentaires).

À n'importe quel moment, on peut faire clic droit dans la fenêtre puis *save* pour sauvegarder tout ce qui a été écrit dans la console (commandes utilisateurs et résultats). C'est assez pratique car sinon l'UI ne nous laisse pas faire de copier-coller...

Note : *dans la suite, on représentera l'invite de* `selt` *avec un tiret (-) comme dans l'outil, et on écrira en dessous ce que l'outil retourne. À noter que pour les exemples de la suite,* `selt` *est appelé sur le réseau de Pétri* `machin` *définit plus haut.*

3.1 Variables de base

Les formules LTL peuvent présenter des "variables", qui sont des références à des *places*. On peut comparer une place à un entier (ou à une autre place), ce qui compare en fait le *nombre de jetons* dans la place :

```
- A >= 5;  La place A contient au moins 5 jetons
TRUE
0.001s
```

On n'est pas obligé de comparer la place ; dans ce cas, `selt` vérifie implicitement que le nombre de jetons dans la place est supérieur ou égal à 1.

```
- B;  La place B a au moins un jeton
TRUE
0.001s
```

Les opérateurs de comparaison disponibles sont :

- `<=` ou `le` : inférieur ou égal
- `>=` ou `ge` : supérieur ou égal
- `lt` : inférieur strictement (lower than)
- `gt` : supérieur strictement (greater than)
- `=` égal

En plus de ça, on a le droit à l'addition (`+`) et à la multiplication (`*`) :

```
- A * 2 + B <= 20 ;
TRUE
0.001s
```

Tina reconnaît aussi un mot "spécial" `dead`, qui qualifie un réseau de Pétri qui ne peut plus progresser dans l'état actuel.

```
- dead ;
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1 ... (preserving - dead)->
  state 1: L.scc*2 A*4 B C
  [accepting all]
0.001s
```

Note : *voir plus bas sur comment interpréter les contre-exemples de* `selt`*.*

3.2 Opérateurs logiques

La LTL présente les opérateurs habituels de la logique :

— `/\`, "et"/conjonction : les propriétés à gauche et droite de l'opérateur sont vraies *en même temps* :

```
- A >= 5 /\ A <= 7 ;  La place A contient au moins 5 jetons et au plus 7
TRUE
0.001s
```

— `\/`, "ou"/disjonction : la propriété à gauche ou celle à droite de l'opérateur (ou les deux) est vraie :

```
- A >= 1 \/ B >= 100 ; La place A contient au moins un jeton ou la place B en contient
                     ; au moins 100 (ou les deux)
TRUE
0.001s
```

— `-`, "non"/négation : la propriété est vraie lorsque la propriété dans le *non* est fausse (et inversement) :

```
- - (A >= 10) ;  La place A ne contient *pas* au moins 10 jetons
TRUE
0.001s
```

— `=>`, "si alors"/implication : lorsque la propriété à gauche est vraie, la propriété à droite doit aussi être vraie :

```
- A => B ; S'il y a un jeton dans A il doit y avoir un jeton dans B
TRUE
0.001s
```

— `<=>`, équivalence : les propriétés à gauche et à droite sont vraies en même temps (si l'une est vraie l'autre est vraie et inversement)

```
- A lt 10 <=> B gt 1 ; Lorsque A a moins de 10 jetons, B en a plus de 1 et inversement
TRUE
0.001s
```

Bien sûr, tout ceci est composable à loisirs :

```
- (A >= 1 \/ B >= 1) /\ (A + B <= 3) \/ dead ;
```

```
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1 ... (preserving - (((A >= 1) \/ B >= 1) /\ (3 >= A + B) \/ dead))->
  state 1: L.scc*2 A*4 B C
  [accepting all]
0.001s
```

3.3 Opérateurs temporels

En plus des opérateurs logiques habituels, la LTL propose deux opérateurs :

— `[]`, toujours : la propriété est *toujours* vraie dans le réseau, peu importe son évolution

```
- [] (A >= 1) ; Il y a *toujours* au moins un jeton dans A
TRUE
0.001s

- [] (A = 7) ; Il y a *toujours* 7 jetons dans A
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1 ... (preserving T)->
  state 1: L.scc*2 A*4 B C
  -t2 ... (preserving - (A = 7))->
  state 2: L.scc*2 A*3 C D
  [accepting all]
0.001s
```

— `<>`, un jour : la propriété *sera vraie* dans le réseau, peu importe son évolution (l'évolution du réseau passe forcément par au moins un état où la propriété est vraie)

```
- <> (C + D gt 0) ;  Un jour le nombre de jetons entre C et D sera supérieur
                  ;  strictement à 0
TRUE
0.001s

- <> C ; Un jour il y aura au moins un jeton dans C (dans tous les cas)
FALSE
  state 0: L.scc*2 A*7 B*3
  -t2 ... (preserving - C)->
* [accepting] state 5: L.dead A*4 D*3
  -L.deadlock ... (preserving - C)->
  state 5: L.dead A*4 D*3
0.001s
```

On peut composer ses opérateurs (entre eux et avec des opérateurs "non-temporels"), par exemple :

— `[] <>` : "à chaque instant, il existe un instant futur où la propriété sera vraie" ; on l'appelle aussi *infiniment souvent* : la propriété n'est pas vraie continûment, mais elle est vraie *régulièrement*, et surtout, peu import jusqu'où on regarde, elle sera toujours vraie régulièrement dans le futur

```
- [] <> D ; Le place D contient infiniment souvent au moins un jeton
TRUE
0.001s
```

— `<> []`, convergence : on atteindra un jour un état où la propriété est vraie, et à ce moment elle continuera d'être vraie indéfiniment

```
- <> [] (A >= 3) ; À partir d'un moment il y aura continûment 3 jetons ou plus dans A
TRUE
0.001s

- [] (D = 2 => (<> [] (D = 3))) ; Si à un moment D contient 2 jetons, alors dans le
                                ; futur elle en contiendra 3 et ce pour toujours
TRUE
0.001s
```

— `- <>`, "non-un jour", jamais : cet opération est un peu l'opposé de `[]` : `- <> P` est vrai si et seulement si `P` n'est *jamais* vrai

```
- - <> (A = 2) ; A ne contiendra jamais 2 jetons
TRUE
0.001s

- - <> (A = 5) ; A ne contiendra jamais 5 jetons
FALSE
  state 0: L.scc*2 A*7 B*3
  -t2 ... (preserving T)->
  state 4: L.scc A*5 B D*2
  -t2 ... (preserving A = 5)->
  state 5: L.dead A*4 D*3
  [accepting all]
0.001s
```

Remarque : en l'absence d'opérateur temporel, la formule donnée à `selt` est vérifiée uniquement pour *l'état initial du réseau*.

```
- A >= 2 ; A contient au moins 2 jetons (à l'état initial)
TRUE
0.001s

- [] (A >= 2) ; A contient au moins 2 jetons (tout le temps)
TRUE
0.001s

- A = 7 ; A contient 7 jetons (à l'état initial)
TRUE
0.001s

- [] (A = 7) ; A contient 7 jetons (tout le temps)
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1 ... (preserving T)->
  state 1: L.scc*2 A*4 B C
  -t2 ... (preserving - (A = 7))->
  state 2: L.scc*2 A*3 C D
  [accepting all]
0.001s
```

3.4 Variables utilisateur

On peut demander à `selt` de retenir une expression complexe dans une variable intermédiaire, à l'aide du mot-clef `op`. Cela permet de réutiliser la variable dans plusieurs requêtes :

```
- op x = A + B ; La "variable" x vaut A + B
operator x : atom
0.000s

- [] (x >= 2) ; Toujours x (= A + B) est supérieur égal à 2
TRUE
0.001s

- <> (x <= 4) ; Un jour x (= A + B) sera inférieur ou égal à 4
TRUE
0.001s
```

Une variable peut contenir une expression (type `atom` comme ci-dessus) ou alors une valeur logique (type `prop`) :

```
- op cd = C \/ D ;
operator cd : prop
0.000s

- [] <> cd ; La place C ou la place D contiennent un jeton infiniment souvent
TRUE
0.001s
```

Les opérateurs peuvent être des fonctions avec des arguments (définition et appel à la Caml) :

```
- op f p x = p \/ C >= x ; f est vraie ssi la propriété p (paramètre) est vraie ou si la
                         ; place C contient au moins x jetons (paramètre)
operator f : prop # atom -> prop
0.000s

- [] <> (f (D gt 1) 1) ; Infiniment souvent, D contient plus d'un jeton, ou C au moins 1
TRUE
0.001s

- [] (f A D) ; Il y a toujours au moins un jeton dans A, ou au moins autant de jeton dans C
             ; que dans D
TRUE
0.001s
```

4 Lancer `selt` en ligne de commande

Il est possible de lancer `selt` directement avec un fichier contenant des phrases LTL, sans passer par l'interface graphique.

Pour cela on construit d'abord le graphe de marquage ou de couverture du réseau de Pétri (ici le réseau est dans le fichier `machin.net`) :

```
> tina machin.net -M machin.ktz
(graphe de marquage)
> tina machin.net -C machin.ktz
(graphe de couverture)
```

Cette commande crée le fichier `machin.ktz` qui contient ledit graphe

Au passage, la commande affiche les propriétés du réseau : bornage, vivacité, nombre d'états et de transition du graphe, états bloquants (dead), états vivants (live).

Ensuite, on peut demander à `selt` de répondre aux questions d'un fichier (ici, `questions.ltl`) sur le graphe de marquage `machin.ktz` :

```
> selt machin.ktz questions.ltl
```

Lorsqu'une formule est fausse, `selt` donne un contre-exemple. On peut l'afficher en détail avec l'option `-p` :

```
> selt machin.ktz -p questions.ltl
```

On peut aussi lui demander simplement les étapes à réaliser pour invalider la formule (= les transitions à franchir) à l'aide de l'option `-s` :

```
> selt machin.ktz -s questions.ltl
```

Plutôt que de passer un fichier entier à `selt`, on peut lui donner une unique formule avec l'option `-f` :

```
> selt machin.ktz -p -f "[] A >= 1"
```

Enfin, on peut aussi lancer `selt` sans fichier et sans formule, auquel cas l'outil devient un invite de commande (comme la version graphique mais dans un terminal), ce qui peut être assez pratique. Les options `-p` et `-s` restent utilisables, et influencent toutes les réponses de `selt` dans la session.

5 Les contre-exemples `selt`

Lorsqu'une propriété sur un réseau est fausse, `selt` nous donne un *contre-exemple*, autrement dit une suite d'actions (= d'événements) à réaliser pour invalider la propriété.

Ce contre-exemple a deux intérêts : si on voulait que la propriété soit vraie, `selt` nous explique pourquoi elle ne l'est pas, ce qui nous permet en quelques sortes de « débugger » le réseau. Si on voulait que la propriété soit fausse, `selt` nous jusitife que c'est bien le cas, et nous donne un scénario qui invalide la propriété.

5.1 Interpréter les contre-exemples

Avec l'option `-p`, `selt` donne les contre-exemples sous la forme d'une suite d'états (= des marquages du réseau) entrecoupés de transitions (= la transition du réseau qui a été franchie).

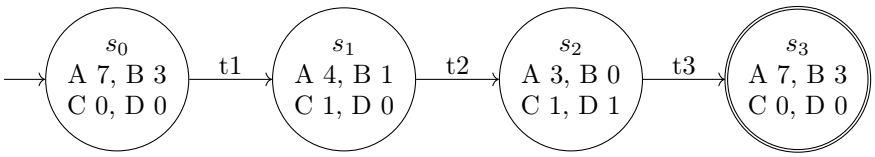
```
- [] B ; Il y a toujours au moins un jeton dans B
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1->
  state 1: L.scc*2 A*4 B C
  -t2->
  state 2: L.scc*2 A*3 C D
  -t3->
  state 3: L.scc*2 A*7 B*3
  [accepting all]
0.001s
```

Un contre-exemple se compose de

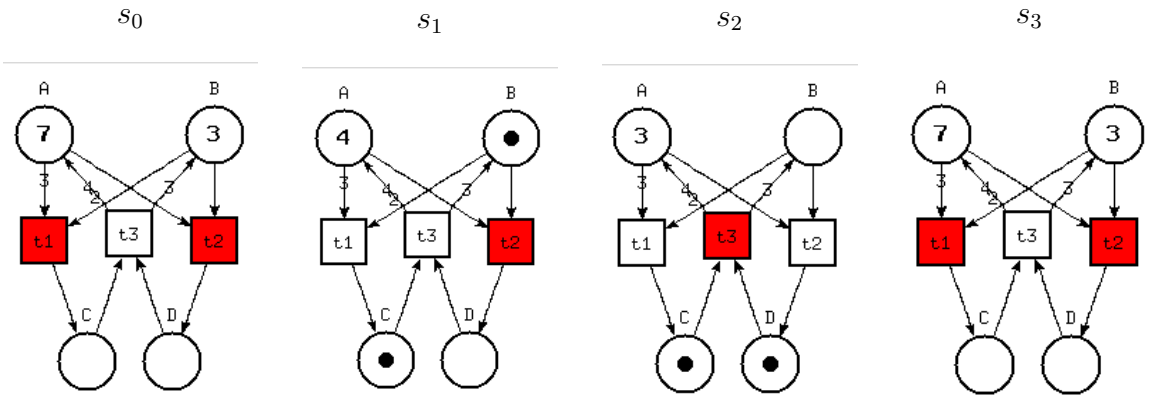
— `state xxx` : numéro de l'état dans lequel on se trouve

— `L.scc*2 A*xx B*yy` : marquage du réseau de Pétri correspondant : `xx` jetons dans A, `yy` jetons dans B, 0 jetons dans toutes les autres places (ignorons `L.scc` qui est une place utilisée par `selt` pour la vivacité)
— `-tx->` : transition qui relie deux états consécutifs (le nom de la transition apparaît ici)
— `[accepting all]` : l'état atteint est valide dans le réseau de Pétri (c'est bien un état atteignable)

Le contre-exemple ci-dessus peut donc s'écrire sous la forme de la *trace* (= succession d'états) suivante :



On peut essayer d'effectuer les actions indiquées dans le stepper, pour voir ce que ça donne :

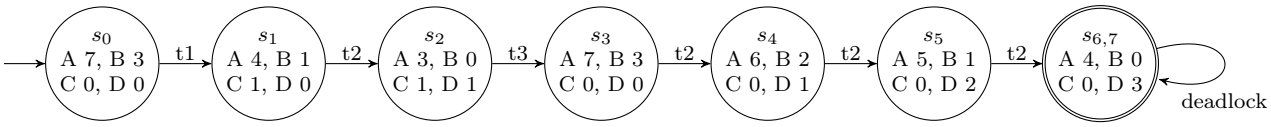


On constate que B = 0 dans l'état s_2, ce qui invalide bien la propriété demandée ($\Box B$).

Prenons une autre propriété plus complexe :

```
- <> [] (B >= 1) ; B a infiniment souvent au moins un jeton
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1->
  state 1: L.scc*2 A*4 B C
  -t2->
  state 2: L.scc*2 A*3 C D
  -t3->
  state 3: L.scc*2 A*7 B*3
  -t2->
  state 4: L.scc*2 A*6 B*2 D
  -t2->
  state 5: L.scc A*5 B D*2
  -t2->
* state 6: L.dead A*4 D*3
  -L.deadlock->
  [accepting] state 7: L.dead A*4 D*3
  -L.deadlock->
  state 6: L.dead A*4 D*3
0.001s
```

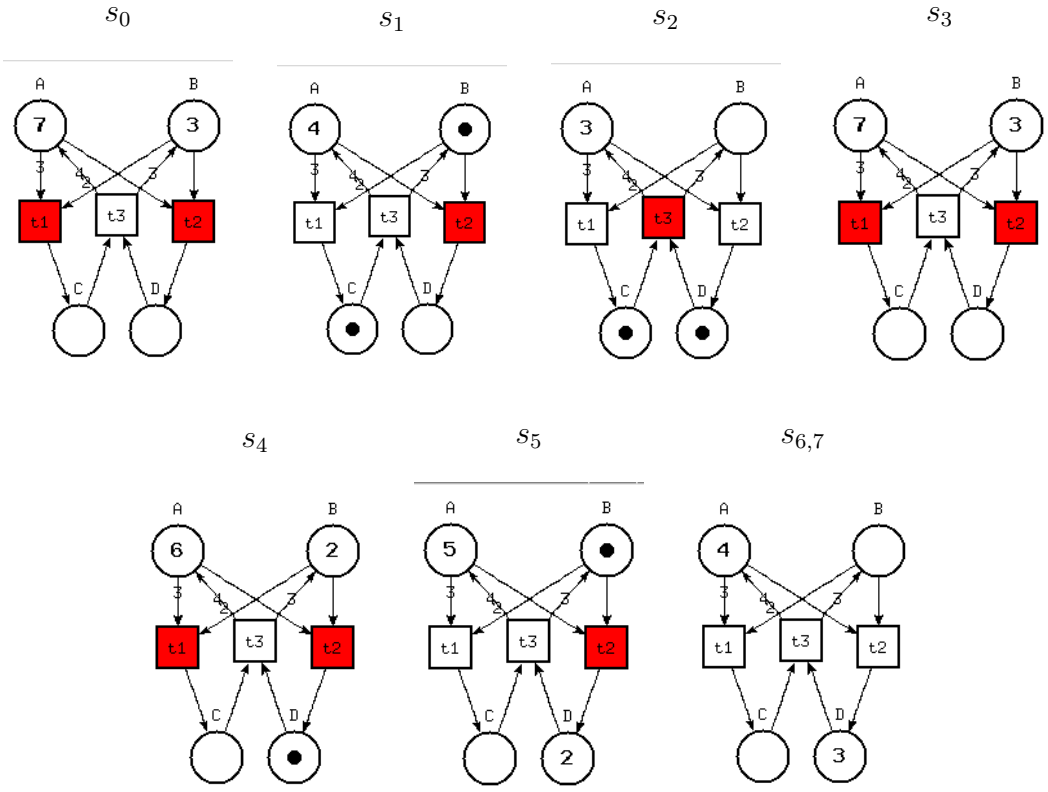
Le contre-exemple ici est beaucoup plus complexe, car il n'est pas suffisant de dire que l'on atteint un état où B a 0 jetons, mais en plus que l'on se trouve dans une situation où B ne contiendra *jamais plus* aucun jeton. Autrement dit, `selt` arrive à nous expliquer comme arriver dans un état où la propriété ne sera jamais plus vraie ; un tel état existe car le système présente un *interblocage* (deadlock), c'est à dire un état duquel on ne peut plus progresser. On peut représenter la trace ainsi :



L'étoile au niveau de `state 6` indique que cet état est le début d'une boucle. La transition spéciale `L.deadlock` indique que l'on ne peut rien faire d'autre que rester dans le même état (l'état `state 6`/`state 7`, qui sont les mêmes).

On constate alors que l'état sur lequel on boucle (A = 4, B = 0, C = 0, D = 3) ne valide pas la propriété $B \geq 1$. Cette propriété, dans cette situation, ne peut donc *pas* être infiniement souvent vraie !

Si l'on essaye ce scénario dans le stepper :



Notez comme dans le dernier état ($s_{6,7}$), aucun transition n'est franchissable (pas de rouge) ; c'est bien que nous sommes en situation d'interblocage.

Si on donne à `selt` l'option `-s` à la place de l'option `-p`, l'outil nous renvoie simplement la liste des transitions à prendre, sans les états intermédiaire :

```
- [] (B = 3) ;
# [] (B = 3 |- FALSE
```

```
t1 t2
# accepting all
0.001s
```

Ici, `selt` nous indique que la propriété est fausse, et qu'il suffit de prendre la transition `t1` puis la transition `t2` pour passer par un état où $B = 3$ est faux.

En fait, ce qui est généré ici est une sorte de « script » pour le stepper. On peut mettre ce script (tout ce qu'il y a après l'invite qui commence par `-` et avant le temps de réponse) dans un fichier `.scn` et le charger dans le stepper avec *File > open history*. Une fois chargé, on peut utiliser les flèches pour avancer ou reculer dans le script.
(À noter que le croisillon indique un commentaire dans les fichiers `.scn`)

On peut demander à `selt` de générer directement le fichier `.scn` (plutôt que de copier coller le résultat) à l'aide de l'option `-S <fichier>` :

```
> selt machin.ktz -f "[] (B = 3)" -S contre-exemple.scn
```

5.2 Contre-exemples et atteignabilité

Lorsque l'on demande si un état est *atteignable* (propriétés du type $\diamond P$), `selt` nous répond soit `FALSE` avec un contre-exemple, soit `TRUE` sans rien d'autre. Notamment, si effectivement on peut atteindre la propriété, `selt` ne nous dit pas *comment* l'atteindre.

La stratégie avec ce genre d'outils est de demander non pas si la propriété est vraie, mais si elle est *fausse*. Comme on s'attend à ce qu'elle soit vraie, demander à `selt` si elle est fausse va générer un contre-exemple, qui dans ce cas est un exemple de comment arriver à un état où la propriété est vraie !

Par exemple, on se demande si on peut atteindre un état où il y a au moins deux jetons dans les places C et D cumulées :

```
- <> (C + D >= 2) ;
TRUE
0.001s
```

L'outil nous répond que oui, sans plus ; mais comment arriver à un tel état ?

On peut lui demander de vérifier que l'état *n'est pas* atteignable, en utilisant la négation. Comme on sait qu'en fait, il l'est, `selt` va nous renvoyer `FALSE` avec un contre-exemple :

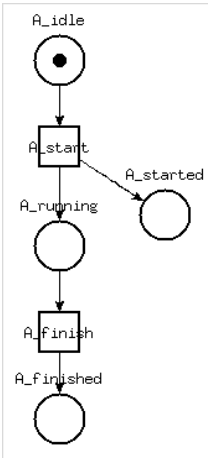
```
- - <> (C + D >= 2) ;
FALSE
  state 0: L.scc*2 A*7 B*3
  -t1->
  state 1: L.scc*2 A*4 B C
  -t2->
  state 2: L.scc*2 A*3 C D
  -t3->
  state 3: L.scc*2 A*7 B*3
  [accepting all]
0.001s
```

Ce contre-exemple est la marche à suivre pour atteindre l'état (le fait qu'on puisse l'atteindre prouve qu'il est atteignable, et donc qu'il n'est pas non-atteignable...). C'est donc un *scénario* qui permet d'atteindre l'état où la propriété est vraie.

Dans le cas présent, on constate que l'état `state 2` a un jeton dans C et un dans D, et donc $C + D \geq 2$.

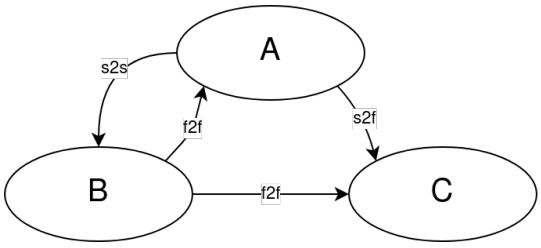
6 Utilisation avec la chaîne d'outils SimplePDL

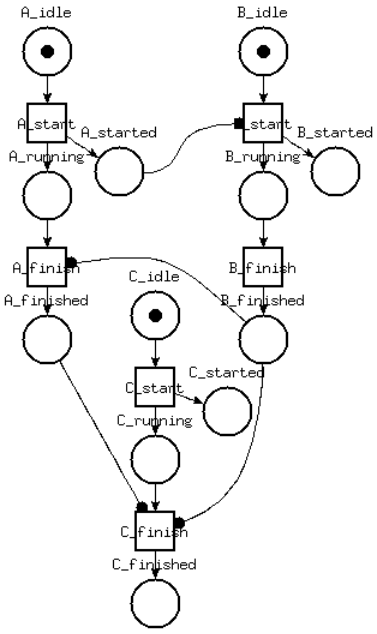
On rappelle qu'un modèle de procédé se transforme en réseau de Pétri. Chaque tâche du procédé se transforme en "sous-réseau de Pétri", par exemple pour une tâche A :



Chaque place correspond à un « état » du réseau : pas encore commencé (*idle*), en cours (*running*), commencé (*started*) et terminé (*finished*).

Bien sûr, un modèle de procédé plus complexe donne un réseau de Pétri plus fourni. Par exemple :





La question qu'on se pose alors c'est : le modèle de procédé termine-t-il *toujours* ? Et aussi, comment démarrer/finir chaque tâche pour qu'il termine ?

On ne peut pas répondre à cette question sur le modèle de procédé directement ; en revanche, on peut aisément y répondre sur le réseau de Pétri !

La stratégie est de générer automatiquement, à partir du modèle SimplePDL en entrée de la transformation, des propriétés LTL qui correspondent à des questions qu'on se pose (terminaison, correction). Il s'agit d'une « simple » transformation de modèle-à-texte ; comme souvent, le plus difficile est de comprendre ce que l'on veut obtenir.

6.1 Terminaison

On considère que le modèle de procédé est terminé lorsque toutes les tâches sont finies. Autrement dit, on veut qu'il y ait un jeton dans toutes les places *finished*.

Pour se simplifier un peu la vie, écrivons une variable qui représente cela :

```
- op finished = (A_finished /\ B_finished /\ C_finished) ;
operator finished : prop
0.000s

- <> finished ; Peut-on toujours atteindre l'état où toutes les tâches son terminées ?
TRUE
0.001s
```

Notre modèle de procédé termine, ouf ! On peut demander à `selt` un exemple de scénario avec le mécanisme présenté dans la Section 5.2 :

```
- - <> finished ;
FALSE
  state 0: L.scc*10 A_idle B_idle C_idle
  -A_start->
  state 1: L.scc*8 A_running A_started B_idle C_idle
  -B_start->
  state 2: L.scc*6 A_running A_started B_running B_started C_idle
  -B_finish->
  state 3: L.scc*4 A_running A_started B_finished B_started C_idle
  -A_finish->
  state 4: L.scc*2 A_finished A_started B_finished B_started C_idle
  -C_start->
  state 5: L.scc A_finished A_started B_finished B_started C_running C_started
  -C_finish->
  state 6: L.dead A_finished A_started B_finished B_started C_finished C_started
  -L.deadlock->
  state 7: L.dead A_finished A_started B_finished B_started C_finished C_started
  [accepting all]
0.001s
```

Si l'on se concentre sur les transitions, on voit qu'on peut :

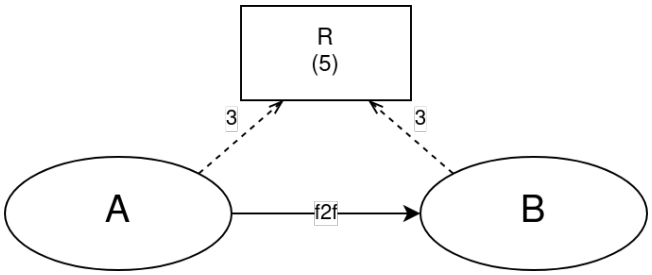
1. démarrer la tâche A (`A_start`)
2. démarrer la tâche B (`B_start`)
3. terminer la tâche B (`B_finish`)
4. terminer la tâche A (`A_finish`)
5. démarrer la tâche C (`C_start`)
6. terminer la tâche C (`C_finish`)

On se retrouve alors dans une situation où les trois tâches sont dans l'état *finished* !

Une dernière chose, on voudrait s'assurer qu'une fois qu'on a fini toutes les tâches, il ne se passe plus rien. On peut, pour cela, exploiter la variable spéciale `dead` :

```
- [] (finished => dead) ; Si on a fini on est en deadlock
TRUE
0.001s
```

Prenons un autre exemple moins trivial, où le partage de ressource peut être un problème :



Demandons à `selt` si on peut toujours atteindre la fin du procédé :

```
- op finished = (A_finished /\ B_finished) ;
operator finished : prop
0.000s

- <> finished ;
FALSE
  state 0: L.scc*5 A_idle B_idle R*5
  -B_start->
* [accepting] state 5: L.scc*4 L.dead A_idle B_running B_started R*2
  -L.deadlock->
  state 5: L.scc*4 L.dead A_idle B_running B_started R*2
0.001s
```

La réponse est non : il est possible de lancer les tâches de façon à ne pas pouvoir les terminer. D'après `selt`, il suffit de commencer la tâche B (`B_start`), et on se retrouve dans un état de d'interblocage. En effet, B ne peut finir que si A a fini, et A ne peut commencer qu'avec 3 ressources, alors qu'il n'en reste que 2 puisque B en a pris 3.

Demandons à `selt` comment terminer ce procédé (on regarde si la propriété *finished* n'est pas atteignable) :

```
- - <> finished ;
FALSE
  state 0: L.scc*5 A_idle B_idle R*5
  -A_start->
  state 1: L.scc*3 A_running A_started B_idle R*2
  -A_finish->
  state 2: L.scc*2 A_finished A_started B_idle R*5
  -B_start->
  state 3: L.scc A_finished A_started B_running B_started R*2
  -B_finish->
  state 4: L.dead A_finished A_started B_finished B_started R*5
  -L.deadlock->
  state 5: L.dead A_finished A_started B_finished B_started R*5
  [accepting all]
0.001s
```

La marche à suivre proposée par l'outil est alors :

1. démarrer la tâche A (`A_start`)
2. terminer la tâche A (`A_finish`)
3. démarrer la tâche B (`B_start`)
4. terminer la tâche B (`B_finish`)

On peut tester ça dans un stepper : ça marche ! On arrive bien à la fin avec un jeton dans toutes les places `finished`.

On remarque que, si on atteint l'état final, on est en deadlock (si on ne l'atteint pas on ne peut pas conclure) :

```
- [] (finished => dead) ;
TRUE
0.001s
```

6.2 Invariants

Les propriétés de terminaison sur les réseaux de Pétri n'ont de conséquence sur les modèles de procédé que si la transformation de l'un à l'autre est *correcte*.

Prouver la correction de la démonstration est un peu difficile ; à la place, on peut vérifier que les propriétés de la sémantique des modèles de procédé est respectée par les réseaux de Pétri résultants. Par exemple :

— chaque tâche X est soit en mode *idle*, soit en mode *running*, soit en mode *finished*

```
- [] (A_idle + A_running + A_finished = 1) ;
TRUE
0.001s
```

— pour chaque tâche X, lorsque cette tâche atteint l'état *started*, il reste dans cet état indéfiniment

```
- [] (A_finished => [] A_finished) ;
TRUE
0.001s
```

(même chose avec l'état *finished*)

— si une ressource R est en quantité n initialement, alors une fois que le procédé est terminé, cette même quantité doit être présente (tout le monde a bien rendu ce qu'il a pris) :

```
- [] (finished => R = 5) ;
TRUE
0.001s
```