TP2 – Raffinement

Notre automate transfert des cartons d'une entrée vers une sortie ; certes, c'est ce qu'il doit faire. Mais ce comportement rudimentaire n'est pas encore suffisamment précis pour pouvoir répondre à des questionnements intéressants sur le système.

En particulier, tel qu'il est écrit, notre automate enlève et dépose des cartons sans que l'on puisse vraiment contrôler où ils se trouvent entre ces deux événements. Par ailleurs, le système ne fait que prendre dans une entrée globale et poser dans une sortie globale, alors que l'on voudrait pouvoir disposer d'un système d'aiguillage des cartons qui soit un peu plus fin.

Par conséquent, nous allons enrichir le formalisme de notre système à l'aide du raffinement. Dans un premier temps, nous nous attaquerons au problème des cartons en transit (ni en entrée ni en sortie) ; ensuite, nous nous attacherons à implémenter un système rudimentaire d'aiguillage de cartons.

1 Un Premier Raffinement

On aimerait pouvoir garder le contrôle des cartons enlevés à l'entrée mais pas encore posés à la sortie. En particulier, cela va permettre de qualifier les cartons "en traitement".

Pour cela, nous allons *raffiner* la machine courante et lui ajouter une variable `Transit`, qui représente les cartons en transit dans l'automate.

Faites un click droit sur la machine (dans l'explorateur de projet) puis choisissez *Refine*. Rodin vous demande un nom pour la machine qui va être créée (pourquoi pas `Automate_1`). L'éditeur s'ouvre alors si une machine très similaire à l'original ; en particulier, elle reprend exactement les mêmes variables et les mêmes événements.

Question 1 : Quel est le type de `Transit` ? Quels invariants, similaires à l'invariant `inv3` de la première machine, permettent de signifier qu'un carton ne peux pas être à la fois en entrée, en transit et en sortie ?

Question 2 : Exprimez sous la forme d'un invariant le fait qu'un carton est toujours en entrée, en transit ou en sortie.

Question 3 : Exprimez sous la forme d'un invariant la propriété suivante : « S'il n'y a pas de carton en entrée ou en transit, alors la sortie contient tous les cartons. »

Ajoutez à la machine la variable `Transit` et les invariants écrits. Nous allons maintenant devoir modifier les événements pour prendre en compte cette nouvelle variable.

On remarquera que les événements vont continuer de faire la même chose ; pour cette raison, nous allons les garder en mode *extended*.

Question 4 : Que contient la variable `Transit` à l'initialisation ? Déduisez-en l'initialisation de cette variable dans l'événement `INITIALISATION`.

Question 5 : Comment évolue `Transit` lorsque l'on prend un carton de l'entrée ? En déduire la ou les actions à ajouter à l'événement `Prendre`. Doit-on modifier les gardes de l'événement ?

Question 6 : Comment évolue `Transit` lorsque l'on pose un carton de la sortie ? En déduire la ou les actions à ajouter à l'événement `Poser`. Doit-on modifier les gardes de l'événement ?

Ajoutez à la machine ces éléments. On pourra tester les différences entre cette machine et l'original à l'aide de ProB.

On pourrait par exemple :

Question 7 : Exprimer en LTL puis vérifier la propriété suivante : « Si il y a des cartons en transit, l'automate finira par déposer un carton sur la sortie. »

2 Système d'Aiguillage

Maintenant que l'on a un meilleur contrôle de la position des cartons dans l'automate, on veut ajouter la possibilité de gérer un aiguillage complexe. Autrement dit, les cartons peuvent avoir plusieurs origines (*arriver* d'endroits différents) et plusieurs destinations (*partir* pour des endroits différents).

Nous allons donc raffiner de nouveau la machine afin de prendre en compte ce mécanisme.

2.1 Études Préliminaires

2.1.1 Arrivées et Départs

Dans notre nouveau modèle, l'automate disposera de plusieurs entrées (des arrivées de cartons en fait) et de plusieurs sorties (des départs de cartons, vers les fameux camions/etc.).

Les entrées et sorties seront identifiées clairement et leur nombre sera fixé. On décide donc de les modéliser sous la forme de deux ensembles constants (ou *carrier sets*) : `ARRIVÉES` et `DÉPARTS`.

Pour chaque carton de l'entrée globale (ou de la sortie globale), il sera possible de connaître son arrivée ou son départ. C'est cette dernière fonctionnalité qui représentera effectivement les arrivées et départs "physique" de la machine.

Comme l'on a concrètement une construction qui à chaque carton associe une arrivée ou un départ, on décide de représenter les lignes d'entrée et de sortie sous la forme de deux fonctions :

Listing 1 – Lignes d'Entrée et de Sortie

```
1 LigneEntrée ∈ CARTONS ⇸ ARRIVÉES
2 LigneSortie ∈ CARTONS ⇸ DÉPARTS
```

Question 8 : Que représente l'opérateur "⇸" ? À quoi correspond en fait le *domaine* de `LigneEntrée` ? Celui de `LigneSortie` ? Sachant que $\mathrm{dom}(f)$ désigne le domaine de f en Event-B, déduisez-en deux prédicats qui relient `Entrée` (resp. `Sortie`) et `LigneEntrée` (resp. `LigneSortie`).

2.1.2 Aiguillage

Afin de pouvoir être aiguillés, les cartons disposent d'un moyen d'être identifié, mais aussi et surtout d'une "adresse", autrement dit de ce qui indique la sortie vers laquelle ils doivent être dirigés. Concrètement, un carton en transit est donc toujours associé à un identifiant de sortie. Une façon intéressante et pratique d'expliciter cela est d'utiliser une structure de donnée dédiée, qui va stocker (entre autres) des couples carton–sortie.

Dans un langage de programmation classique, on utiliserait des enregistrements ou des objets, mais on n'a pas accès à ce genre de chose en Event-B (ce qui est un choix, rappelons-le).

Néanmoins, il existe une façon standard de représenter ce genre de structures avec en plus les avantages d'Event-B (c'est-à-dire le raffinement et la correction) : une famille de fonctions, qui représente chacune un champ de l'enregistrement (et prennent en entrée ledit enregistrement).

Par exemple, pour représenter un carton en transit, on définit un ensemble abstrait `TRANSITIONS`, dont les éléments sont ces fameux enregistrement (qui ne sont pas entièrement spécifiés). On a ensuite deux fonctions qui permettent d'accéder aux champs qui nous intéressent :

Listing 2 – "Enregistrement" `TRANSITIONS`

```
1 carton_de_t ∈ TRANSITIONS ⇸ CARTONS
2 départ_de_t ∈ TRANSITIONS ⇸ DÉPARTS
```

On stocke alors les transitions en cours dans une variable `Transitions` $\subseteq$ `TRANSITIONS`.

Question 9 : Quelle propriété doivent avoir les fonctions `carton_de_t` et `départ_de_t` pour qu'elles soient toujours bien définies pour tout carton en transit ?

La structure actuelle est largement suffisante en terme de modélisation, mais dans l'idéal, on aimerait qu'elle se substitue à `Transit`. On notera qu'il est possible en Event-B d'exprimer l'*image* ou *codomaine* d'une fonction f avec la formule $\text{ran}(f)$.

Question 10 : Qu'est-ce qui relie `Transit` et `Transitions` ? *(On exprimera cette relation sous forme d'un prédicat.)*

Question 11 : Comment exprimer le fait que chaque transition est bien reliée à un et un seul carton ?

2.2 Développement Event-B

La phase de questions préliminaires étant finie, on peut maintenant passer à l'écriture du raffinement.

2.2.1 Extension du Contexte

Faites un click droit sur le contexte déjà écrit (dans l'explorateur de projets) puis clickez sur *Extend* (on vous demandera un nom pour le contexte, pourquoi pas `Automate_ctx_2`). Rodin va alors créer le contexte et lui faire étendre le précédent !

Dans ce nouveau contexte, on ajoutera trois nouveaux ensembles nécessaires au développement (voir listing 3).

Listing 3 – Contexte Étendu

```
1 CONTEXT
2     Automate_ctx_2
3 EXTENDS
4     Automate_ctx_0
5 SETS
6     ARRIVÉES    -- Des lignes d'arrivée vers l'automate
7     DÉPARTS     -- Des lignes de départ depuis l'automate
8     TRANSITIONS -- Objets possibles représentant des transitions
9 END
```

2.2.2 Raffinement de la Machine

Initier, comme dans la section précédente, un raffinement de la machine `Automate_1` (appelons-la `Automate_2`).

Attention à bien changer le `SEES` pour qu'il fasse référence au bon contexte.

Comme décrit précédemment, nous allons substituer les variables abstraites `Entrée`, `Sortie` et `Transit` par des variables concrètes `LigneEntrée`, `LigneSortie` et `Transitions`. Nous allons ensuite ajouter les variables `carton_de_t` et `départ_de_t`.

Le début de la machine ressemble donc au code donné au listing 4.

Listing 4 – Début de la Machine Raffinée

```
1  MACHINE
2      Automate_2
3  REFINES
4      Automate_1
5  SEES
6      Automate_ctx_2
7  VARIABLES
8      Transitions     -- Variables du système
9      LigneEntrée
10     LigneSortie
11     carton_de_t     -- Champs accessibles d'une transition
```

```
12      départ_de_t
13  INVARIANTS
14      inv1: LigneEntrée ∈ CARTONS ⇸ ARRIVÉES
15      inv2: LigneSortie ∈ CARTONS ⇸ DÉPARTS
16      inv3: Transitions ⊆ TRANSITIONS
17      inv4: carton_de_t ∈ Transitions ⤖ CARTONS
18      inv5: départ_de_t ∈ Transitions → DÉPARTS
```

Question 12 : Complétez les invariants avec ceux exprimés dans les questions 8 (`inv6` et `inv7`) et 10 (`inv8`). Quel est le rôle de ces invariants ?

2.2.3 Initialisation

Par défaut, les événements raffinés générés lors de la création de la machine sont en mode "extended" ; cela signifie qu'ils font au moins exactement ce que la machine abstraite fait (d'où les lignes en grisé).

Cependant, comme nous avons remplacé les variables initiales (`Entrée`, `Transit` et `Sortie`) au profit d'autres expressions, nous allons devoir supprimer les lignes précédentes. Pour cela, il faut d'abord clicker sur le "extended" et ainsi passer en mode "not extended".

Question 13 : Sachant que, initialement, il n'y a pas de cartons en sortie, comment initialiseriez-vous la variable `LigneSortie` ? *(N'oubliez pas les fonctions sont des ensembles.)*

Question 14 : De même, sachant que, initialement, il n'y a pas de cartons en transit, comment initialiser `Transitions`, `carton_de_t` et `départ_de_t` ?

Il ne reste plus qu'à initialiser `LigneEntrée`. Naïvement, on pourrait penser qu'on puisse mettre cette variable à $\emptyset$, mais cela est incompatible avec l'invariant qui relie cette variable à `Entrée`, qui est initialisée à `CARTONS` dans la machine abstraite.

Une autre démarche (par ailleurs tout à fait valide) consisterait à créer une constante `LigneEntrée0` dans le contexte à laquelle initialiser la variable et qui aurait les bons axiomes pour respecter les invariants ; mais il existe une autre façon de faire que nous allons privilégier ici : l'*affectation non-déterministe*.

Question 15 : Expliquez ce qu'est l'affectation non-déterministe et ses différentes formes. Quelle forme privilégieriez-vous dans ce cas précis ? Écrivez alors l'affectation de `LigneEntrée` de façon à respecter tous les invariants.

2.2.4 Événement `Prendre`

Tout comme pour `INITIALISATION`, l'événement `Prendre` n'est pas "extended" ; opérez le changement.

L'événement initial n'avait pour unique paramètre que le carton à prendre ; mais dans notre modèle actuel, les cartons viennent de quelque part et vont quelque part.

Question 16 : Complétez le modèle pour pouvoir prendre en compte l'arrivée et le départ du carton en train d'être pris. *(N'oubliez pas les gardes correspondante !)*

Question 17 : On remarquera que la garde `grd2` n'a plus vraiment de sens étant donné que `Entrée` a disparu. Par quoi devrait-on la remplacer ?

Question 18 : Quelle garde devrait-on ajouter (en plus du type des paramètres bien sûr) pour s'assurer que le carton que l'on prend était bien dans l'arrivée demandée ?

L'événement commence donc ainsi :

Listing 5 – Début de `Prendre`

```
1  Prendre <not extended>
2  REFINES
3      Prendre
4  ANY
```

```
5       c
6       a
7       d
8   WHERE
9       grd1: c ∈ CARTONS    -- "Type" de c
10      grd2: c ∈ ...        -- Contrainte sur c (question 17)
11      grd3: a ∈ ARRIVÉES   -- "Type" de a
12      grd4: ...            -- Contrainte sur c et a (question 18)
13      grd5: d ∈ DÉPARTS    -- "Type" de d
```

La sémantique des *event parameters* est un peu différente de celle des paramètres d'une fonction "classique" : un événement ne peux pas vraiment "retourner" de valeur ; on va plutôt considérer que la valeur existe depuis toujours et que l'événement se contente de la configurer et de la prendre en compte.

On va donc ajouter un paramètre `t` de type `TRANSITIONS` (l'ensemble des transitions qui existent). On doit par ailleurs s'assurer que cette transition n'a pas déjà été créée (autrement dit que le carton n'est pas *déjà* en transit) :

Listing 6 – Gardes sur `t`

```
1   Prendre <not extended>
2   ...
3   ANY
4       ...
5       t
6   WHERE
7       ...
8       grd6: t ∈ TRANSITIONS
9       grd7: t ∉ Transitions
```

Dans les actions de l'événement, on va tout d'abord retirer la ligne qui concerne `Transit` et la remplacer par des actions sur `t` et `Transitions`. Concrètement, on va "configurer" `t` puis l'ajouter à l'ensemble `Transitions` :

Listing 7 – Actions sur `t`

```
1       act1: carton_de_t(t) := c
2       act2: départ_de_t(t) := d
3       act3: Transitions := Transitions ∪ { t }
```

Autre subtilité : il faut encore supprimer la ligne concernant `Entrée` pour la remplacer par un traitement sur `LigneEntrée` ; autrement dit, il faut modéliser le fait que l'on prenne le carton de la ligne d'entrée.

> **Question 19 :** Comment se traduit (informellement) le fait de "retirer un carton de" `LigneEntrée` ? Quel opérateur d'Event-B permet justement de faire une telle chose ? Écrivez alors l'action qui correspond à cela. *(Gardez à l'esprit que* ***`LigneEntrée`*** *est une fonction.)*

2.2.5 Événement `Poser`

Encore une fois, on commence par passer en mode "not extended".

> **Question 20 :** De quoi a-t-on besoin pour pouvoir réaliser cet événement ? Où se trouvent ces informations ? Écrivez alors le début de l'événement `Poser`, et plus particulièrement son ou ses paramètres et ses gardes. *(Garder à l'esprit que cet événement est responsable de poser le carton au bon endroit.)*

Dans les actions de l'événement, il faudra remplacer les références à `c` en `carton_de_t(c)` (qui correspond au carton en cours de transit que l'on est en train de traiter).

Il va falloir par ailleurs réécrire l'action `act1` (qui retire le carton à `Transit`). Concrètement, cela va surtout consister à retirer `t` de `Transitions`.

Question 21 : L'action décrite ci-dessus respecte-t-elle bien tous les invariants ? Si non, que faire pour que ce soit le cas ?

Dernière chose, la ligne qui concerne `Sortie` n'a plus lieu d'être. Il va falloir la remplacer, et ainsi modéliser le fait que l'on ait posé le carton sur une ligne de sortie.

Question 22 : Comment exprimeriez-vous en Event-B que le carton est sur une ligne de sortie spécifique ? Écrivez l'action correspondante.

Une Dernière Chose...

Si vous sauvegardez votre machine actuellement (dans l'éventualité où elle ne contiendrait aucune erreur), vous allez sans doute remarquer que Rodin souligne en jaune l'événement `Poser`.

En passant votre souris sur le panneau *attention* (⚠), Rodin vous explique que *"Witness for `c` missing. Default witness generated"*.

Question 23 : Qu'est-ce qu'un *witness* (ou *témoin*) ? Pourquoi Event-B en attend-il un dans ce contexte ? Écrivez le witness relatif à `c`. *(Le label d'un témoin en Event-B doit être le nom de la variable correspondante.)*