

TP1 – Des Cartons et des Hommes

La société *Ad Hoc Logistics* (pour Automatic Distribution of Higher Order Carton) est spécialisée dans la fabrication et l'acheminement de cartons. Dans ce procédé, elle emploie un grand nombre d'employés pour déplacer les cartons de la sortie de la chaîne de fabrication aux camions qui se chargeront du transport.

Seulement voilà, ce déplacement manuel a du mal à tenir la demande croissante en cartons et cause parfois des erreurs, l'être humain étant faillible.

Aussi, *Ad Hoc* a décidé d'investir dans leur centre de R&D dans le but de réaliser un automate logistique qui se chargera de réaliser le travail qui incombait alors aux humains et dont le fonctionnement serait **partiellement prouvé** à l'aide d'Event-B.

En tant que spécialiste des cartons et de la logistique (mais aussi d'Event-B), votre rôle va donc être de concevoir et de prouver cet automate.

1 Abstraction

Le principe à la base d'un automate logistique est de conduire des objets d'un point à un autre ; mais la façon de déplacer ces objets pourrait très bien changer sans que le fonctionnement global du système ne soit remis en cause. Pour cette raison, il apparaît intéressant de se pencher, dans un premier temps, sur une *abstraction* du système ; autrement dit, une représentation très générale et très générique de ce dernier.

Question 1 : Devra-t-on modéliser chaque type d'automate que l'on voudra implémenter ? Est-il pertinent de modéliser (à ce niveau) le mode de transport des cartons ? Le contenu des cartons ? Les cartons ? Dressez alors une liste exhaustive et minimale de ce dont on a besoin pour écrire ce modèle abstrait. Donnez ensuite les opérations les plus basiques dont devrait être capable le système.

Le but du modèle abstrait est en fait de définir les "bonnes" propriétés générales que devraient avoir un automate quel qu'il soit. En particulier, on peut s'intéresser à deux propriétés d'un tel automate :

1. L'automate ne perd pas de carton
2. L'automate finit par traiter tous les cartons

Dans un premier temps, on s'intéresse à la première propriété.

Question 2 : Exprimez la propriété (1) avec la logique d'Event-B (premier ordre et théorie des ensembles). De quel genre de propriété s'agit-il ?

2 Modèle Event-B

(Note : les notations proposées le sont à titre indicatif. Libre à vous de concevoir votre modèle comme bon vous semble.)


Maintenant que nous avons mené notre réflexion préliminaire sur la conception de ce système, nous allons pouvoir passer à l'écriture de son modèle Event-B.

Ouvrez le logiciel *Rodin*¹ et créez un nouveau projet (*AutomateLogistique* par exemple).

Notez que les éditeurs de *Rodin* sont des *sémantiques* : on ajoute des champs et des parties à un fichier via un menu contextuel (click-droit sur un élément pour afficher ce menu) ; c'est seulement une fois créés que vous pourrez éditer les champs du modèle.

1. Sur les machines de l'école, Rodin se trouve à `/mnt/n7fs/ens/rodin/rodin/rodin`

2.1 Contexte

Créez un nouveau composant Event-B de type *contexte* en cliquant sur  dans l'explorateur. *Rodin* vous demande un nom (`Automate_ctx_0` par exemple).

Ce contexte va principalement contenir un *ensemble abstrait* (*carrier set*) : l'ensemble des cartons qui seront traités par la machine. On écrit alors :

Listing 1 – Première Partie du Contexte


```

1  CONTEXT
2      Automate_ctx_0
3  SETS
4      CARTONS          — Ensemble des cartons
5  END

```

Complétez le contexte dans *Rodin* à l'aide du menu contextuel : click-droit dans le contexte puis *Add Carrier Set*.

2.2 Début de la Machine

Le contexte étant maintenant terminé, on peut créer une machine qui verra ce contexte et modélisera la partie dynamique de notre automate. Toujours dans le même projet, créez un nouveau composant Event-B avec , cette fois-ci de type *machine* (que l'on pourra appeler `Automate_0`).

L'automate abstrait possède trois variables : son entrée, sa sortie, et les cartons qu'il est en train de déplacer. Ces trois variables sont des ensembles de cartons. On écrit donc :

Listing 2 – Début de la Machine

```

1  MACHINE
2      Automate_0
3  SEES
4      Automate_ctx_0
5  VARIABLES
6      Entrée          — Variables du système
7      Sortie
8  INVARIANTS
9      inv1: Entrée ⊆ CARTONS — "Type" des variables
10     inv2: Sortie ⊆ CARTONS

```

Notez que l'on pourrait écrire également $\text{Entrée} \in \mathbb{P}(\text{CARTONS})$, ce qui est tout à fait équivalent (en tout cas d'un point de vue modélisation).

Complétez la machine dans *Rodin* (toujours à l'aide du click-droit dans la machine).

Histoire de préparer le terrain pour de futures preuves, il est nécessaire de remarquer une propriété intéressante du système : un carton ne peut pas être à la fois en entrée et en sortie.

Question 3 : Écrivez l'invariant auquel cela correspond (`inv3`).

2.3 Événements

Une machine Event-B commence toujours par un événement appelé **INITIALISATION**, dans la machine par défaut. Cet événement sert à initialiser les différentes variables, autrement dit **Entrée** et **Sortie** dans notre cas.

Question 4 : Que contiennent les ensembles **Entrée** et **Sortie** au tout début du fonctionnement de la machine ? Déduisez-en les actions à écrire dans l'événement **INITIALISATION**.

Complétez l'événement **INITIALISATION** dans *Rodin* (click-droit sur l'événement puis *Add Action*).

Comme déterminé plus haut, l'automate logistique fait fondamentalement deux choses : prendre des cartons de l'entrée et poser des cartons à la sortie, ce que l'on va modéliser par deux événements dans la machine (pourquoi pas **Prendre** et **Poser**).

Il est à noter que ce n'est pas l'automate qui "choisit" quel carton il prend ou pose. Généralement, le carton arrive ou est là, et l'automate fait avec. Pour modéliser cette subtilité, Event-B met à disposition un concept qui peut justement être interprété de cette façon : les paramètres d'événements ou *event parameters* (champs **ANY**).

Le début des événements s'écrit donc comme ceci :

Listing 3 – Débuts de **Prendre** et **Poser**

```

1  Prendre
2  ANY
3      c
4  WHERE
5      grd1: c ∈ CARTONS      — "Type" de c
6      grd2: ...              — Contrainte sur c (cf q. 5)
7  THEN
8      act1: Entrée := ...     — Retirer c à l'entrée (cf q. 7)
9  END
10
11 Poser
12 ANY
13     c
14 WHERE
15     grd1: c ∈ CARTONS      — "Type" de c
16     grd2: ...              — Contraintes sur c (cf q. 6)
17 THEN
18     act1: Sortie := ...     — Ajouter c à la sortie
19 END

```

Ajoutez et complétez ces événements dans *Rodin* (à l'aide de click-droit + *Add Event*).

Question 5 : Où l'automate peut-il prendre un carton ? Déduisez-en la garde **grd2** de l'événement **Prendre**.

Question 6 : À quelle(s) condition(s) un carton peut-il être déposé sur la sortie ? Déduisez-en la garde **grd2** de l'événement **Poser**. (*On prendra soin de ne pas déposer un carton qui est déjà sur la sortie, ni un carton qui n'a pas encore été pris par l'automate.*)

Pour écrire les actions de ces événements, il ne faut pas oublier que les variables que l'on traite sont des ensembles.

Question 7 : Comment écrire le fait de retirer ou d'ajouter un carton à l'entrée et à la sortie ? Déduisez-en les actions à écrire pour **Prendre** et **Poser**. (*Attention à bien identifier ce qui est un ensemble et ce qui est un élément de cet ensemble.*)

Questions subsidiaires

Nous nous attacherons plus tard aux obligations de preuve générées par ce modèle. Néanmoins, il est des questions que l'on peut déjà se poser :

Question 8 : Les affectations faites dans l'événement **INITIALISATION** respectent-elles bien tous les invariants ?

Question 9 : Qu'est-ce qui garantit que des opérations telles que \setminus (différence ensembliste) et \cup (union ensembliste) ne changent pas la nature des ensembles utilisés dans le modèle ? (Autrement dit que les ensembles restent des ensembles de cartons.)

Question 10 : Donnez un argument (informel) au fait que des cartons ne se dupliquent pas. Est-il possible que des cartons disparaissent ?

3 Animation de Modèle

Avant de passer à la suite, parlons un peu d'animation de modèle et de *model checking*.

Faites un click droit sur la machine et cliquez ensuite sur *Start Animation / Model Checking*. Rodin va ouvrir la perspective *ProB*.

ProB est un model-checker basé sur Prolog et qui permet d'une part d'animer des modèles Event-B en simulant les événements de la machine, et d'autre part de vérifier certaines propriétés à base de model-checking (typiquement, des propriétés LTL sur la machine).

3.1 Simulation Manuelle

En haut à gauche de la fenêtre se trouve la liste des événements, incluant un événement spécial *SETUP_CONTEXT*, qui permet d'initialiser ce qui doit l'être (donc ici : **CARTONS**).

Double-cliquez sur cet événement : ProB va initialiser **CARTONS** avec des objets. Double-cliquez sur *INITIALISATION* pour réaliser l'initialisation de la machine. Vous aurez alors accès à un ensemble **Entrée** contenant un carton. Vous pouvez alors cliquer sur **Prendre** puis sur **Poser...** et c'est à peu près tout !

Cliquez sur l'icône 🔄 pour recommencer la simulation. Cette fois-ci, faite un click droit sur *SETUP_CONTEXT* et sélectionnez *Non-deterministic choice #2*. Cela va initialiser **CARTONS** avec deux objets.

Double-cliquez sur les différents événements pour les réaliser. Lorsque plusieurs façons de réaliser un événement sont possibles, vous pouvez faire un click-droit et sélectionner un choix (malheureusement les choix ne sont pas très explicites).

3.2 Violation d'Invariant

La fonctionnalité d'animation de modèle de ProB est utile lorsqu'il s'agit de se convaincre sommairement que le système n'a pas de comportement aberrant ; mais la vraie force de ce outil réside dans ses capacités de vérification et de recherche de contre-exemples.

Par exemple, ProB permet de chercher les violations d'invariants par la machine : cliquez sur la flèche à côté de *Checks* et choisissez *Model Checking*. Dans la boîte de dialogue, ne cochez que la case *Find Invariant Violations* puis cliquez sur *Start Model Checking*. Si vous avez bien écrit la machine, ProB devrait terminer sans message d'erreur.

Pour tester cette fonctionnalité plus avant, vous pouvez volontairement écrire un invariant faux et voir, avec la même technique, si ProB le détecte.

3.3 Propriétés LTL

En plus de ce que nous avons vu, ProB est capable de vérifier des propriétés LTL (Logique Temporelle Linéaire) à l'aide de *model checking*. Pour cela, vous pouvez cliquer à nouveau sur la flèche à côté de *Checks* et sélectionner *LTL Model Checking*. ProB vous demande alors de rentrer une formule.

La syntaxe de ProB pour la LTL est partiellement décrite dans la table 2. Vous pouvez également la retrouver à cette adresse https://www3.hhu.de/stups/prob/index.php/LTL_Model_Checking.

Symbole	Description
G	<i>Globally</i> , toujours, \square
F	<i>Finally</i> , éventuellement, \diamond
{ ... }	Tester un prédicat écrit en B
&	Et logique
or	Ou logique
=>	Implication logique
[Op]	Teste si le prochain événement à être exécuté est Op

TABLE 1 – Éléments de Syntaxe LTL ProB

Par exemple, on peut exprimer en LTL que l'entrée de l'automate n'est jamais vide avec

$$G \{ \text{Entrée} \neq \{\} \} \quad \text{ou encore} \quad G \{ \text{card}(\text{Entrée}) > 0 \}$$

Si vous demandez à ProB de vérifier cette formule, il vous répondra qu'il a trouvé un contre-exemple à la propriété écrite et vous propose de vous la montrer. Vous obtiendrez alors quelque chose de similaire à la figure 6.

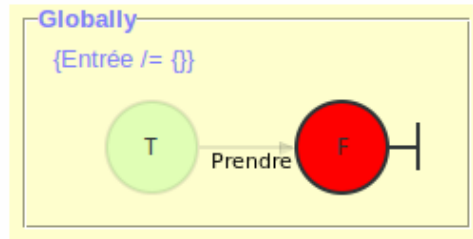


FIGURE 1 – Contre-exemple de ProB

Concrètement, cela vous indique qu'une façon de violer la formule est de partir d'un état initial avec une entrée qui contient un carton puis de faire une transition **Prendre** (ce qui vide bien la variable **Entrée**).

Question 11 : Exprimez en LTL puis vérifiez la propriété (2) du début du TP. (*Deux solutions équivalentes pour cette question.*)

Question 12 : Comment exprimer le fait qu'un carton ait été pris de l'entrée mais pas encore acheminé vers la sortie? Exprimez alors en LTL puis vérifiez la propriété suivante : « À tout moment, s'il y a des cartons pris mais pas encore acheminés, l'automate finira par poser un carton. ».

Question 13 : Exprimez en LTL puis vérifiez la propriété de non-duplication énoncée dans la question 10.

Question 14 : Exprimez en LTL puis vérifiez la propriété de conservation énoncée dans la question 10.

TP2 – Raffinement

Notre automate transfère 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 peut 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 4 – 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 $\text{dom}(f)$ 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'explicitier 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 5 – "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 cliquez 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 6).

Listing 6 – 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 7.

Listing 7 – 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 cliquer 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 8 – 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 peut 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 9 – 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 10 – 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 *témoignage*) ? 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.*)

TP3 – Des Cartons de Preuves

En Event-B (comme dans toutes les approches formelles par ailleurs), un **modèle n'est rien sans preuve**.

Le modèle que l'on a créé décrit en effet un automate logistique comme prévu, mais tant que l'on n'aura pas réalisé de preuves dessus, on ne pourra rien en tirer d'intéressant.

Fort heureusement, Rodin génère automatiquement les obligations de preuve à décharger sur le modèle pour garantir sa cohérence. La plupart du temps, les obligations de preuve concernent les invariants; autrement dit, il s'agira de prouver que les événements ne violent pas les invariants donnés dans la machine.

1 Preuves de Modèle




Dans l'explorateur de projets, dépliez l'élément correspondant à la première machine (**Automate_1**). Vous devez alors apercevoir un sous-item appelé *Proof Obligations*; dépliez cet élément pour obtenir la liste des obligations de preuve générées par Rodin.



Notez que les obligations de preuves les plus simples (typiquement concernant l'initialisation, somme toute assez simple) ont déjà été déchargées automatiquement : elles sont en vert et ont un petit "A" sur leur icône. A contrario, les obligations de preuve en orange n'ont pas encore été déchargées, et il va donc nous incomber de le faire.



Commencez par basculer en perspective "Proving" (Window > Perspective > Open Perspective > Other..., puis sélectionner "Proving"). Double-cliquez ensuite sur une preuve qu'il reste à faire, par exemple l'obligation de preuve **Prendre/INV/inv3** (une preuve que l'INVARIANT **inv3** n'est pas violé par l'événement **Prendre**). Vous devriez tomber sur une vue similaire à celle donnée dans la figure 4.


La fenêtre est alors coupée en trois grandes zones :

- L'arbre de preuve pour l'obligation de preuve en cours (1), qui retrace toutes les étapes prises dans la preuve
- L'explorateur de projet (2) avec notamment la liste des obligations de preuve
- Le prouveur lui même (3), avec en haut la liste des hypothèses, au milieu le but (ce qui doit être prouvé) et en bas l'assistant

Pour réaliser une preuve, on peut s'y prendre de deux manières : les preuves ou buts très simples peuvent être complétées ou au moins partiellement avancées avec les prouveurs automatiques tels que PP, p0, p1, ml ou encore l'icône  (qui applique automatiquement un certain nombre de tactiques) et  (pour "nettoyer" un peu les hypothèses et la conclusion), ou même  (qui appelle des solveurs SMT). Les autres preuves quant à elles doivent être réalisées "interactivement".

Une preuve interactive consiste principalement à cliquer sur les opérateurs en rouge (ce qui indique qu'il est possible de transformer l'expression), ajouter des hypothèses en l'écrivant dans la zone de texte en bas et en appuyant sur  (pour *add hypothesis*), ou encore en utilisant des règles d'inférence comme "ct" (pour *contradict*). Si des hypothèses semblent manquantes, il est par ailleurs possible d'aller les chercher automatiquement avec le "lasso" .

Enfin, il est possible de remonter dans un arbre de preuve avec , ou même revenir à un noeud précis et détruire tout ce qui se trouve en-dessous avec .

Pour cette preuve très simple, placez vous à la racine et utilisez  (si la preuve était déjà faite). Cliquez sur le signe égal dans le but.

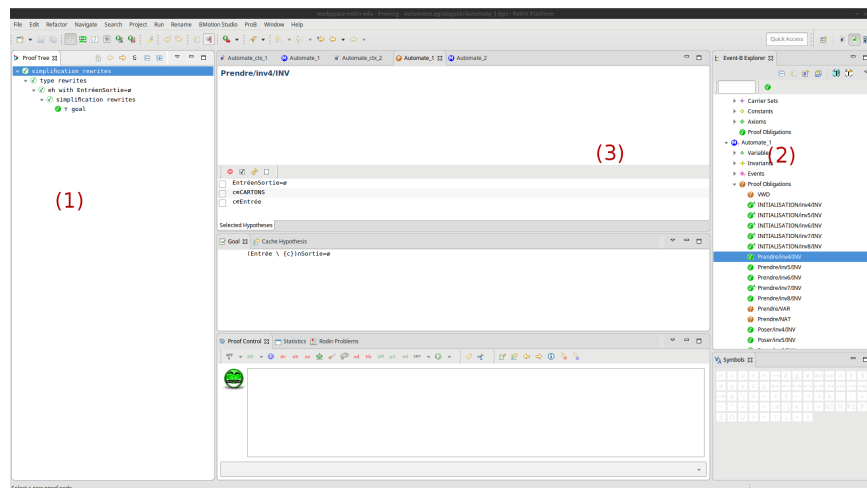


FIGURE 2 – Vue "Proving" de Rodin

Le prouveur transforme le but en deux nouveaux buts, l'un avec à la place le signe \subseteq et l'autre avec le signe \supseteq (notez que ce second but a été prouvé automatiquement car il est très simple).

Clickez alors sur le symbole \subseteq et choisissez "Remove inclusion" pour réécrire le but sans le symbole d'inclusion.

Le prouveur va modifier encore le but (avec quelques étapes intermédiaires) et vous propose maintenant de prouver que $x = c$ sachant que $x \in \text{Entrée} \cap \text{Sortie}$ et $\text{Entrée} \cap \text{Sortie} = \emptyset \dots$ Surveillez le signe égal dans les hypothèses et choisissez "*Apply equality from left to right*".

Cela a pour effet de remplacer la première occurrence du membre de gauche de l'égalité (**Entrée** \cap **Sortie**) avec son membre de droite (\emptyset). Autrement dit, le prouveur va transformer $x \in \text{Entrée} \cap \text{Sortie}$ en $x \in \emptyset$ et détecter tout seul qu'il y a une contradiction, ce qui fait la preuve !

À titre indicatif, l'arbre de preuve :

$$\frac{\frac{\frac{\frac{\frac{\frac{E \cap S = \emptyset, c \in E, \perp \vdash x = c}{E \cap S = \emptyset, c \in E, x \in \emptyset \vdash x = c}}{E \cap S = \emptyset, c \in E, x \in E \cap S \vdash x = c}}{E \cap S = \emptyset, c \in E \vdash x \in E \cap S \Rightarrow x = c}}{E \cap S = \emptyset, c \in E \vdash \forall x \cdot x \in E \cap S \Rightarrow x = c}}{E \cap S = \emptyset, c \in E \vdash \forall x \cdot x \in E \cap S \Rightarrow x \in \{c\}}}{\frac{E \cap S = \emptyset, c \in E \vdash \emptyset \subseteq (E \setminus \{c\}) \cap S}{E \cap S = \emptyset, c \in E \vdash (E \setminus \{c\}) \cap S = \emptyset}}$$

FIGURE 3 – Arbre de Preuve pour Prendre/inv4/INV

Question 1 : Déchargez les autres obligations de preuve de la première machine du mieux que vous pouvez.

Question 2 : Déchargez les obligations de preuve de la deuxième machine.

Note : les obligations de preuve de la troisième machine sont un peu plus techniques... Ne planchez dessus que si vous avez le temps !

2 Autres Propriétés

(Dans cette section, on s'intéresse de nouveau à la première et à la deuxième machine.)

Tout au début du sujet, nous avons énoncé une propriété qui pourrait nous intéresser sur nos automates logistiques, la propriété (2).

Question 3 : De quel genre de propriété s'agit-il ?

Contrairement à la propriété (1), il est difficile d'exprimer (2) sous forme d'un prédicat. For heureusement, nous disposons d'un outil parfaitement adapté à cela : le *variant*.

Question 4 : Qu'est-ce qu'un variant ? Dans quel cas très simple (que vous avez très certainement déjà rencontré) l'utilise-t-on ?

Pour rappel, la propriété disait : « l'automate finit par traiter tous les cartons ». Malheureusement, dans la première machine, il est impossible de qualifier tous les cartons (et notamment les cartons pris mais pas encore posés) ; nous allons donc d'abord nous intéresser à une propriété analogue mais plus simple : « l'automate finit par prendre tous les cartons de l'entrée ».

En supposant que tout carton pris finit par être déposé, on remarquera que ces deux propositions sont relativement équivalentes.

Question 5 : Parmi les actions des événements, qu'est-ce qui modélise en particulier le fait de prendre un carton de l'entrée ? Quel donnée dans le modèle peut alors servir de variant ? (*Rappel : un variant est une variable strictement décroissante.*)

Un seul événement fait décroître le variant : **Prendre**. Pour signifier à Rodin que c'est à cet événement que l'on s'intéresse, on va le marquer *convergent*. Pour cela, cliquez sur *ordinary* puis, dans le menu déroulant, sélectionnez *convergent*. Cela permet d'indiquer à Rodin que l'événement va s'effectuer un nombre fini de fois.

Une fois que c'est fait, ajoutez à la machine le fameux variant. Notez que de nouvelles obligations de preuve apparaissent : **FIN**, qui représente le fait que le variant ne décroît pas indéfiniment, et **Prendre/VAR**, qui vérifie que l'événement convergent fait bien décroître strictement le variant.

Note : avant de continuer, remarquez qu'un variant ne peut décroître finiment que si il est effectivement fini. Dans le cas présent, cela revient à dire que l'ensemble des cartons est fini. Afin de pouvoir réaliser la preuve **FIN**, nous allons donc ajouter l'axiome *finite(CARTONS)* dans le contexte de la machine.

Question 6 : Réalisez la preuve **FIN**. (*Pour prouver qu'un ensemble est fini, on peut écrire dans la zone de texte un sur-ensemble de cet ensemble dont on sait qu'il est fini puis cliquer sur l'opérateur finite.*)

Note : il est possible de prendre pour variant ou bien un ensemble (pour lequel la relation d'ordre strict est \subset) ou alors le cardinal de cet ensemble (avec pour relation d'ordre $<$). Notez que, si ces deux représentations sont équivalentes, les preuves pour la seconde sont un peu plus délicates car elles font appel à l'arithmétique.

Pour aller plus loin

Le variant que nous avons écrit nous assure que l'automate peut finir par prendre tous les cartons ; mais cela ne suffit pas à dire qu'il finira par *traiter* tous les cartons. En effet, pour cela, il va falloir s'intéresser à la deuxième machine, et notamment à la variable **Transit**.

On aimerait écrire un variant pour **Entrée** et un autre pour **Transit** ; mais le problème est qu'une machine ne peut avoir qu'un seul variant. Il va donc falloir trouver un bon variant qui concerne à la fois ces deux variables.

Question 7 : Une approche naïve consiste à considérer l'expression **Entrée** \cup **Transit** ou $\text{card}(\text{Entrée}) + \text{card}(\text{Transit})$. Pourquoi ces variants ne conviennent-ils pas ?

Question 8 : Proposez un variant pour cette machine.

Avec le nouveau variant, on remarquera que **Prendre** et **Poser** sont convergents.

On remarquera par ailleurs que, parmi les nouvelles obligations de preuve, on a : **VWD** (pour **Variant Well Definedness**) qui vérifie si le variant est correctement défini, autrement dit qu'il utilise les opérateurs qu'il a le droit d'utiliser, ainsi que **Prendre/NAT** (pour **NATural**), qui vérifie que le variant reste bien toujours positif.

Note : les preuves pour ce variant sont techniques – en grande partie à cause de l'ergonomie du prouveur. Vous pouvez tenter de faire les parties faciles puis vous convaincre (ou prouver à la main) que les obligations de preuve sont bien correctes.

Remarquez que ce nouveau variant signifie que, éventuellement, l'entrée et l'automate seront vides. En conjonction avec l'invariant $Entrée = \emptyset \wedge Transit = \emptyset \Rightarrow Sortie = CARTONS$ écrit dans un TP précédent, cela signifie par ailleurs que la sortie finira par contenir tous les cartons que l'automate devait traiter.

3 Réflexions et Conclusion

Question 9 : La compagnie *Ad Hoc Logistics* décide de réaliser un automate avec 1 ligne d'entrée et un nombre quelconque de lignes de sortie. Comment feriez-vous pour fournir un modèle de cette machine ?

Question 10 : La compagnie veut maintenant précisément un automate avec 1 ligne d'entrée et 5 lignes de sortie. Comment feriez-vous pour fournir un modèle de cette machine ?

Question 11 : Quel avantage voyez-vous à avoir réalisé des modèles abstraits d'automate ?

TP – Développement Avec Event-B

Partie 1 – Des Cartons et des Hommes

La société *Ad Hoc Logistics* (pour Automatic Distribution of Higher Order Carton) est spécialisée dans la fabrication et l'acheminement de cartons. Dans ce procédé, elle emploie un grand nombre d'employés pour déplacer les cartons de la sortie de la chaîne de fabrication aux camions qui se chargeront du transport.

Seulement voilà, ce déplacement manuel a du mal à tenir la demande croissante en cartons et cause parfois des erreurs, l'être humain étant faillible.

Aussi, *Ad Hoc* a décidé d'investir dans leur centre de R&D dans le but de réaliser un automate logistique qui se chargera de réaliser le travail qui incombait alors aux humains et dont le fonctionnement serait **partiellement prouvé** à l'aide d'Event-B.

En tant que spécialiste des cartons et de la logistique (mais aussi d'Event-B), votre rôle va donc être de concevoir et de prouver cet automate.

1 Abstraction

Le principe à la base d'un automate logistique est de conduire des objets d'un point à un autre ; mais la façon de déplacer ces objets pourrait très bien changer sans que le fonctionnement global du système ne soit remis en cause. Pour cette raison, il apparaît intéressant de se pencher, dans un premier temps, sur une *abstraction* du système ; autrement dit, une représentation très générale et très générique de ce dernier.

Question 1 : Devra-t-on modéliser chaque type d'automate que l'on voudra implémenter ? Est-il pertinent de modéliser (à ce niveau) le mode de transport des cartons ? Le contenu des cartons ? Les cartons ? Dressez alors une liste exhaustive et minimale de ce dont on a besoin pour écrire ce modèle abstrait. Donnez ensuite les opérations les plus basiques dont devrait être capable le système.

Le but du modèle abstrait est en fait de définir les "bonnes" propriétés générales que devraient avoir un automate quel qu'il soit. En particulier, on peut s'intéresser à deux propriétés d'un tel automate :

1. L'automate ne perd pas de carton
2. L'automate finit par traiter tous les cartons

Dans un premier temps, on s'intéresse à la première propriété.

Question 2 : Exprimez la propriété (1) avec la logique d'Event-B (premier ordre et théorie des ensembles). De quel genre de propriété s'agit-il ?

2 Modèle Event-B


(Note : les notations proposées le sont à titre indicatif. Libre à vous de concevoir votre modèle comme bon vous semble.)

Maintenant que nous avons mené notre réflexion préliminaire sur la conception de ce système, nous allons pouvoir passer à l'écriture de son modèle Event-B.

Ouvrez le logiciel *Rodin*¹ et créez un nouveau projet (**AutomateLogistique** par exemple). Pour éditer un fichier, nous recommandons très très fortement l'utilisation du *Rodin Editor* (click droit sur un fichier, "Open With" puis "Rodin Editor").

Notez que les éditeurs de *Rodin* sont des *sémantiques* : on ajoute des champs et des parties à un fichier via un menu contextuel (click-droit sur un élément pour afficher ce menu) ; c'est seulement une fois créés que vous pourrez éditer les champs du modèle.

2.1 Contexte

Créez un nouveau composant Event-B de type *contexte* en cliquant sur  dans l'explorateur. *Rodin* vous demande un nom (**Automate_ctx_0** par exemple).

Ce contexte va principalement contenir une *ensemble abstrait* (*carrier set*) : l'ensemble des cartons qui seront traités par la machine. On écrit alors :

Listing 11 – Première Partie du Contexte


```

1  CONTEXT
2      Automate_ctx_0
3  SETS
4      CARTONS                -- Ensemble des cartons
5  END

```

Complétez le contexte dans *Rodin* à l'aide du menu contextuel : click-droit dans le contexte puis *Add Carrier Set*.

2.2 Début de la Machine

Le contexte étant maintenant terminé, on peut créer une machine qui verra ce contexte et modélisera la partie dynamique de notre automate. Toujours dans le même projet, créez un nouveau composant Event-B avec , cette fois-ci de type *machine* (que l'on pourra appeler **Automate_0**).

L'automate abstrait possède trois variables : son entrée, sa sortie, et les cartons qu'il est en train de déplacer. Ces trois variables sont des ensembles de cartons. On écrit donc :

Listing 12 – Début de la Machine

```

1  MACHINE
2      Automate_0
3  SEES
4      Automate_ctx_0
5  VARIABLES
6      Entrée                -- Variables du système
7      Sortie
8  INVARIANTS
9      inv1: Entrée ⊆ CARTONS -- "Type" des variables
10     inv2: Sortie ⊆ CARTONS

```

Notez que l'on pourrait écrire également $\text{Entrée} \in \mathbb{P}(\text{CARTONS})$, ce qui est tout à fait équivalent (en tout cas d'un point de vue modélisation).

Complétez la machine dans *Rodin* (toujours à l'aide du click-droit dans la machine).

Histoire de préparer le terrain pour de futures preuves, il est nécessaire de remarquer une propriété intéressante du système : un carton ne peut pas être à la fois en entrée et en sortie.

Question 3 : Écrivez l'invariant auquel cela correspond (*inv3*).

1. Sur les machines de l'école, Rodin se trouve à `/mnt/n7fs/ens/rodin/rodin/rodin`

2.3 Événements

Une machine Event-B commence toujours par un événement appelé **INITIALISATION**, dans la machine par défaut. Cet événement sert à initialiser les différentes variables, autrement dit **Entrée** et **Sortie** dans notre cas.

Question 4 : Que contiennent les ensembles **Entrée** et **Sortie** au tout début du fonctionnement de la machine ? Déduisez-en les actions à écrire dans l'événement **INITIALISATION**.

Complétez l'événement **INITIALISATION** dans *Rodin* (click-droit sur l'événement puis *Add Action*).

Comme déterminé plus haut, l'automate logistique fait fondamentalement deux choses : prendre des cartons de l'entrée et poser des cartons à la sortie, ce que l'on va modéliser par deux événements dans la machine (pourquoi pas **Prendre** et **Poser**).

Il est à noter que ce n'est pas l'automate qui "choisit" quel carton il prend ou pose. Généralement, le carton arrive ou est là, et l'automate fait avec. Pour modéliser cette subtilité, Event-B met à disposition un concept qui peut justement être interprété de cette façon : les paramètres d'événements ou *event parameters* (champs **ANY**).

Le début des événements s'écrit donc comme ceci :

Listing 13 – Débuts de **Prendre** et **Poser**

```

1  Prendre
2  ANY
3      c
4  WHERE
5      grd1: c ∈ CARTONS          — "Type" de c
6      grd2: ...                  — Contrainte sur c (cf q. 5)
7  THEN
8      act1: Entrée := ...        — Retirer c à l'entrée (cf q. 7)
9  END
10
11 Poser
12 ANY
13     c
14 WHERE
15     grd1: c ∈ CARTONS          — "Type" de c
16     grd2: ...                  — Contraintes sur c (cf q. 6)
17 THEN
18     act1: Sortie := ...        — Ajouter c à la sortie
19 END

```

Ajoutez et complétez ces événements dans *Rodin* (à l'aide de click-droit + *Add Event*).

Question 5 : Où l'automate peut-il prendre un carton ? Déduisez-en la garde **grd2** de l'événement **Prendre**.

Question 6 : À quelle(s) condition(s) un carton peut-il être déposé sur la sortie ? Déduisez-en la garde **grd2** de l'événement **Poser**. (*On prendra soin de ne pas déposer un carton qui est déjà sur la sortie, ni un carton qui n'a pas encore été pris par l'automate.*)

Pour écrire les actions de ces événements, il ne faut pas oublier que les variables que l'on traite sont des ensembles.

Question 7 : Comment écrire le fait de retirer ou d'ajouter un carton à l'entrée et à la sortie ? Déduisez-en les actions à écrire pour **Prendre** et **Poser**. (*Attention à bien identifier ce qui est un ensemble et ce qui est un élément de cet ensemble.*)

Questions subsidiaires

Nous ne nous attacherons pas aux obligations de preuve générées par ce modèle dans le cadre de ce TP. Néanmoins, il est des questions que l'on peut se poser :

Question 8 : Les affectations faites dans l'événement `INITIALISATION` respectent-elles bien tous les invariants ?

Question 9 : Qu'est-ce qui garantit que des opérations telles que \setminus (différence ensembliste) et \cup (union ensembliste) ne changent pas la nature des ensembles utilisés dans le modèle ? (Autrement dit que les ensembles restent des ensembles de cartons.)

Question 10 : Donnez un argument (informel) au fait que des cartons ne se dupliquent pas. Est-il possible que des cartons disparaissent ?

Partie 2 – Raffinement

Notre automate transfère 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.

3 Réalisation du 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 clic 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 11 : Quel est le type de `Transit` ? Quels invariants, similaires à l'invariant `inv3` de la première machine, permettent de signifier qu'un carton ne peut pas être à la fois en entrée, en transit et en sortie ?

Question 12 : Exprimez sous la forme d'un invariant le fait qu'un carton est toujours en entrée, en transit ou en sortie.

Question 13 : 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 14 : Que contient la variable `Transit` à l'initialisation ? Déduisez-en l'initialisation de cette variable dans l'événement `INITIALISATION`.

Question 15 : 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 16 : 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.

Partie 3 – Animation et Preuve (Optionnel)

Dans les parties précédents, nous nous sommes attachés à écrire des modèles complets et *informellement* corrects. Dans la pratique, un modèle Event-B seul **n'a aucune valeur sans preuve**. Dit autrement, le but fondamental de la méthode Event-B, c'est non seulement de formaliser des systèmes de manière mathématique, mais aussi et surtout de *vérifier* ces modèles, à l'aide d'animation (model-checking) et de preuve.

Dans cette partie, nous allons voir comment l'on peut utiliser Rodin afin d'*animer* (simuler, "exécuter") un modèle Event-B. Dans un second temps, nous regarderons un peu les preuves que l'on peut faire sur des modèles Event-B, à l'aide du prouveur de Rodin.

4 Animation de Modèle

Parlons tout d'abord un peu d'animation de modèle et de *model checking*.

Faites un click droit sur la machine et cliquez ensuite sur *Start Animation / Model Checking*. Rodin va ouvrir la perspective *ProB*.

ProB est un model-checker basé sur Prolog et qui permet d'une part d'animer des modèles Event-B en simulant les événements de la machine, et d'autre part de vérifier certaines propriétés à base de model-checking (par exemple, la violation d'invariant).

4.1 Simulation Manuelle

En haut à gauche de la fenêtre se trouve la liste des événements, incluant un événement spécial *SETUP_CONTEXT*, qui permet d'initialiser ce qui doit l'être (donc ici : **CARTONS**).

Double-cliquez sur cet événement : ProB va initialiser **CARTONS** avec des objets. Double-cliquez sur *INITIALISATION* pour réaliser l'initialisation de la machine. Vous aurez alors accès à un ensemble **Entrée** contenant *un* carton. Vous pouvez alors cliquer sur **Prendre** puis sur **Poser...** et c'est à peu près tout !

Cliquez sur l'icône 🔄 pour recommencer la simulation. Cette fois-ci, faite un click droit sur *SETUP_CONTEXT* et sélectionnez *Non-deterministic choice #2*. Cela va initialiser **CARTONS** avec deux objets.

Double-cliquez sur les différents événements pour les réaliser. Lorsque plusieurs façons de réaliser un événement sont possibles, vous pouvez faire un click-droit et sélectionner un choix (malheureusement les choix ne sont pas très explicites).

4.2 Violation d'Invariant

La fonctionnalité d'animation de modèle de ProB est utile lorsqu'il s'agit de se convaincre sommairement que le système n'a pas de comportement aberrant ; mais la vraie force de ce outil réside dans ses capacités de vérification et de recherche de contre-exemples.

Par exemple, ProB permet de chercher les violations d'invariants par la machine : cliquez sur la flèche à côté de *Checks* et choisissez *Model Checking*. Dans la boîte de dialogue, ne cochez que la case *Find Invariant Violations* puis cliquez sur *Start Model Checking*. Si vous avez bien écrit la machine, ProB devrait terminer sans message d'erreur.

Pour tester cette fonctionnalité plus avant, vous pouvez volontairement écrire un invariant faux et voir, avec la même technique, si ProB le détecte.

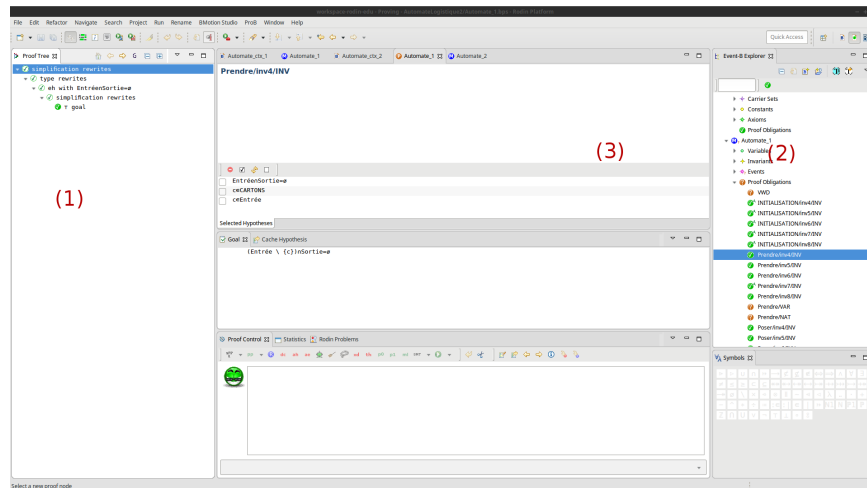


FIGURE 4 – Vue "Proving" de Rodin

5 Preuves de Modèle

Pour tout modèle Event-B, Rodin génère automatiquement un ensemble d'obligations de preuve, qu'il est nécessaire de décharger (= prouver) afin de garantir la cohérence dudit modèle. La plupart du temps, les obligations de preuve concernent les invariants ; autrement dit, il s'agira de prouver que les événements ne violent pas les invariants donnés dans la machine.



Dans l'explorateur de projets, déployez l'élément correspondant à la première machine (**Automate_1**). Vous devez alors apercevoir un sous-item appelé *Proof Obligations* ; déployez cet élément pour obtenir la liste des obligations de preuve générées par Rodin.

Notez que les obligations de preuves les plus simples (typiquement concernant l'initialisation, somme toute assez simple) ont déjà été déchargées automatiquement : elles sont en vert et ont un petit "A" sur leur icône. A contrario, les obligations de preuve en orange n'ont pas encore été déchargées, et il va donc nous incomber de le faire.


Commencez par basculer en perspective "Proving" (Window > Perspective > Open Perspective > Other..., puis sélectionner "Proving"). Double-cliquez ensuite sur une preuve qu'il reste à faire, par exemple l'obligation de preuve **Prendre/INV/inv3** (une preuve que l'INVARIANT **inv3** n'est pas violé par l'événement **Prendre**). Vous devriez tomber sur une vue similaire à celle donnée dans la figure 4.



La fenêtre est alors coupée en trois grandes zones :


- L'arbre de preuve pour l'obligation de preuve en cours (1), qui retrace toutes les étapes prises dans la preuve
- L'explorateur de projet (2) avec notamment la liste des obligations de preuve
- Le prouveur lui même (3), avec en haut la liste des hypothèses, au milieu le but (ce qui doit être prouvé) et en bas l'assistant

Pour réaliser une preuve, on peut s'y prendre de deux manières : les preuves ou buts très simples peuvent être complétées ou au moins partiellement avancées avec les prouveurs automatiques tels que PP, p0, p1, ml ou encore l'icône  (qui applique automatiquement un certain nombre de tactiques) et  (pour "nettoyer" un peu les hypothèses et la conclusion), ou même **SMT** (qui appelle des solveurs SMT). Les autres preuves quant à elles doivent être réalisées "interactivement".

Une preuve interactive consiste principalement à cliquer sur les opérateurs en rouge (ce qui indique qu'il est possible de transformer l'expression), ajouter des hypothèses en l'écrivant dans la zone de texte en bas et en appuyant sur **ah** (pour *add hypothesis*), ou encore en utilisant des règles d'inférence comme "ct"

(pour *contradict*). Si des hypothèses semblent manquantes, il est par ailleurs possible d'aller les chercher automatiquement avec le "lasso" .

Enfin, il est possible de remonter dans un arbre de preuve avec , ou même revenir à un noeud précis et détruire tout ce qui se trouve en-dessous avec .

Pour cette preuve très simple, placez vous à la racine et utilisez  (si la preuve était déjà faite). Cliquez sur le signe égal dans le but.

Le prouveur transforme le but en deux nouveaux buts, l'un avec à la place le signe \subseteq et l'autre avec le signe \supseteq (notez que ce second but a été prouvé automatiquement car il est très simple).

Cliquez alors sur le symbole \subseteq et choisissez "Remove inclusion" pour réécrire le but sans le symbole d'inclusion.

Le prouveur va modifier encore le but (avec quelques étapes intermédiaires) et vous propose maintenant de prouver que $x = c$ sachant que $x \in \text{Entrée} \cap \text{Sortie}$ et $\text{Entrée} \cap \text{Sortie} = \emptyset$... Surveillez le signe égal dans les hypothèses et choisissez "Apply equality from left to right".

Cela a pour effet de remplacer la première occurrence du membre de gauche de l'égalité ($\text{Entrée} \cap \text{Sortie}$) avec son membre de droite (\emptyset). Autrement dit, le prouveur va transformer $x \in \text{Entrée} \cap \text{Sortie}$ en $x \in \emptyset$ et détecter tout seul qu'il y a une contradiction, ce qui fait la preuve !

À titre indicatif, l'arbre de preuve :

$$\begin{array}{c}
 \frac{\cdot}{E \cap S = \emptyset, c \in E, \perp \vdash x = c} \\
 \frac{E \cap S = \emptyset, c \in E, \perp \vdash x = c}{E \cap S = \emptyset, c \in E, x \in \emptyset \vdash x = c} \\
 \frac{E \cap S = \emptyset, c \in E, x \in \emptyset \vdash x = c}{E \cap S = \emptyset, c \in E, x \in E \cap S \vdash x = c} \\
 \frac{E \cap S = \emptyset, c \in E \vdash x \in E \cap S \Rightarrow x = c}{E \cap S = \emptyset, c \in E \vdash \forall x \cdot x \in E \cap S \Rightarrow x = c} \\
 \frac{E \cap S = \emptyset, c \in E \vdash \forall x \cdot x \in E \cap S \Rightarrow x \in \{c\}}{E \cap S = \emptyset, c \in E \vdash (E \setminus \{c\}) \cap S \subseteq \emptyset} \\
 \frac{E \cap S = \emptyset, c \in E \vdash \emptyset \subseteq (E \setminus \{c\}) \cap S}{E \cap S = \emptyset, c \in E \vdash (E \setminus \{c\}) \cap S = \emptyset}
 \end{array}$$

FIGURE 5 – Arbre de Preuve pour **Prendre/inv4/INV**

Question 17 : Déchargez les autres obligations de preuve de la première machine du mieux que vous pouvez.

Question 18 : Déchargez les obligations de preuve de la deuxième machine.

Partie 4 – Model Checking Avancé & Propriétés LTL (Optionnel)

En plus de ce que nous avons vu, ProB est capable de vérifier des propriétés LTL (Logique Temporelle Linéaire) à l'aide de *model checking*. Pour cela, vous pouvez cliquer à nouveau sur la flèche à côté de *Checks* et sélectionner *LTL Model Checking*. ProB vous demande alors de rentrer une formule.

La syntaxe de ProB pour la LTL est partiellement décrite dans la table 2. Vous pouvez également la retrouver à cette adresse https://www3.hhu.de/stups/prob/index.php/LTL_Model_Checking.

Symbole	Description
G	<i>Globally</i> , toujours, \square
F	<i>Finally</i> , éventuellement, \diamond
{ ... }	Tester un prédicat écrit en B
&	Et logique
or	Ou logique
=>	Implication logique
[Op]	Teste si le prochain événement à être exécuté est Op

TABLE 2 – Éléments de Syntaxe LTL ProB

Par exemple, on peut exprimer en LTL que l'entrée de l'automate n'est jamais vide avec

$$G \{ \text{Entrée} \neq \{\} \} \quad \text{ou encore} \quad G \{ \text{card}(\text{Entrée}) > 0 \}$$

Si vous demandez à ProB de vérifier cette formule, il vous répondra qu'il a trouvé un contre-exemple à la propriété écrite et vous propose de vous la montrer. Vous obtiendrez alors quelque chose de similaire à la figure 6.

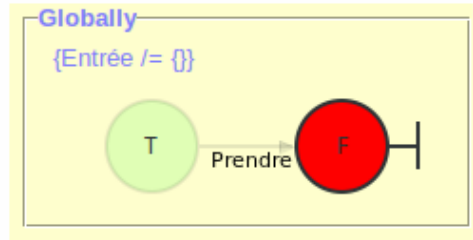


FIGURE 6 – Contre-exemple de ProB

Concrètement, cela vous indique qu'une façon de violer la formule est de partir d'un état initial avec une entrée qui contient un carton puis de faire une transition **Prendre** (ce qui vide bien la variable **Entrée**).

Question 19 : Exprimez en LTL puis vérifiez la propriété (2) du début du TP. (*Deux solutions équivalentes pour cette question.*)

Question 20 : Comment exprimer le fait qu'un carton ait été pris de l'entrée mais pas encore acheminé vers la sortie? Exprimez alors en LTL puis vérifiez la propriété suivante : « À tout moment, s'il y a des cartons pris mais pas encore acheminés, l'automate finira par poser un carton. ».

Question 21 : Exprimez en LTL puis vérifiez la propriété de non-duplication énoncée dans la question 10.

Question 22 : Exprimez en LTL puis vérifiez la propriété de conservation énoncée dans la question 10.