

Finite State Machine

Rapport

Machine à café



Titouan Le Mao
Julien N'Diaye

Année 2019-2020

Sommaire

| | |
|--|-----------|
| Mise en contexte | 3 |
| Fonctionnalités de notre machine à café | 4 |
| Produit minimum viable | 4 |
| Options | 5 |
| Exemple d'options | 5 |
| Gestion des ingrédients | 6 |
| Fidélisation NFC | 6 |
| Extensions | 7 |
| Gestion de la soupe | 7 |
| Gestion de l'iced tea | 7 |
| Gestion de l'avancement de la préparation | 8 |
| Détection des gobelets | 8 |
| Version finale | 9 |
| Choix de conception | 10 |
| Conception de la machine à état fini | 10 |
| Orthogonal state | 10 |
| Gestion du temps | 11 |
| Machine sur écoute | 11 |
| Séparation des systèmes de paiement | 11 |
| Implémentation d'une interface | 12 |
| Utilisation de threads | 12 |
| Choix des classes | 14 |
| Activité de vérification et validation par Itsa | 15 |
| Propriétés LTL | 15 |
| Sécurité | 15 |
| Vérifications additionnelles | 16 |
| Prise de recul | 17 |

Mise en contexte

De nos jours, la consommation de boissons chaudes comme le café ne cesse d'augmenter. Les goûts sont de plus en plus variés et la méthode de conception des boissons chaudes intéresse de plus en plus de monde.

Les clients de machines à café fournissent de plus en plus d'admonestation aux machines à café professionnelles, s'en suit logiquement un nombre d'exigences de plus en plus grand, complexe et diversifié.

Notre machine à café apporte une réponse à tous ces problèmes et se situe à l'opposé de la négligence utilisateur. Elle assure la sécurité de l'utilisateur lors de la préparation de boissons chaudes ou froides, permet de choisir différents types de boissons avec plusieurs préparations différentes et a une interface simple et intuitive.

Nous allons donc vous présenter à travers ce rapport, les choix de conceptions et les implémentations que nous avons réalisés sur notre machine à café.

Fonctionnalités de notre machine à café

1) Produit minimum viable

Tout d'abord, nous avons développé un produit minimum viable qui assure le bon fonctionnement d'une machine à café d'un point de vue utilisateur; c'est-à-dire pouvoir choisir son type de café grâce à des boutons, payer par pièce de 10, 20, 50 centimes ou par carte bleu puis récupérer son café.

Ce produit minimum viable permet d'avoir 3 types de boissons :

- Café
- Espresso
- Thé

Notre machine à café affiche le prix des boissons une fois que nous les avons sélectionnées. Le changement de sélection de boisson est possible avant l'encaissement de l'argent.

Elle dispose également de 3 sliders qui permettent de choisir la température, la taille et le degré de sucre d'une boisson.

Le slider sucre permet 5 positions, le slider size permet 3 tailles de boissons et le slider température permet de choisir entre 4 températures différentes.

Si aucune action utilisateur n'est effectuée dans les 45 secondes suivantes, les sliders reviennent à un état initial et la sélection est annulée. Les sliders reviennent aussi à un état initial dès lors qu'une transaction est effectuée,

Le paiement peut s'effectuer avant ou après la sélection de la boisson. Si le paiement est effectué et qu'il n'y a aucune action utilisateur les 45 secondes suivantes, le paiement est abandonné et l'argent est rendu si l'utilisateur a payé par pièces.

Un utilisateur peut annuler la préparation de sa boisson avant la préparation de cette dernière et est remboursé s'il a payé par pièces. Si le paiement est effectué par pièce et que son montant est supérieur au prix de la boisson, la monnaie est rendue.

Lors de la préparation d'une recette, aucune action ne peut être réalisée.

Enfin, une fois la préparation terminée et la boisson récupérée, la machine entre en séquence de nettoyage. À la suite de cela, elle revient dans son état initial et peut de nouveau préparer de nouvelles boissons.

Options

Notre MVP permet également à l'utilisateur de choisir un certain nombre d'options en fonction de la boisson sélectionnée. Nous avons choisi de les représenter par des *Checkbox* que l'utilisateur sélectionne en fonction de ses désirs. Toutes les options exigées ont été ajoutées à notre machine et les recettes auxquelles elles appartiennent ont également été modifiées en conséquence.

Options

☐ Nuage de lait (+0.10€)

☐ Sirop d'érable (+0.10€)

☐ Glace vanille mixée (+0.60€)

Options

☐ Croutons (+0.30€)

Exemple d'options

Lorsque la préparation d'une boisson est en cours, les *Checkbox* sont désactivées pour éviter que l'utilisateur les changent pendant la préparation (voir ci-dessous).

Options

☐ Nuage de lait (+0.10€)

☐ Sirop d'érable (+0.10€)

☒ Glace vanille mixée (+0.60€)

Enfin, la machine affiche lorsqu'une option est indisponible à cause d'un manque d'ingrédient.

Options

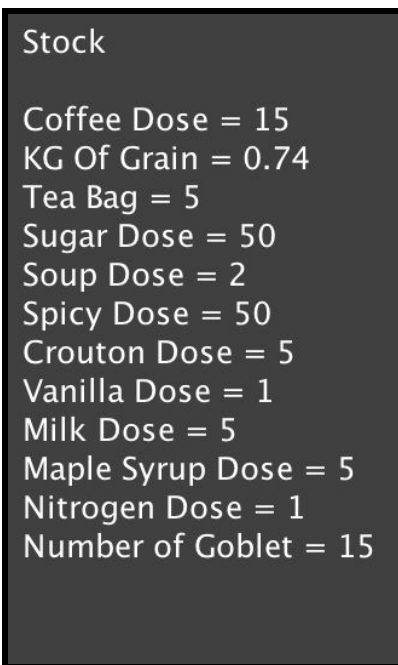
☐ Nuage de lait (+0.10€)

☐ Sirop d'érable (+0.10€)

Glace vanille indisponible.

Gestion des ingrédients

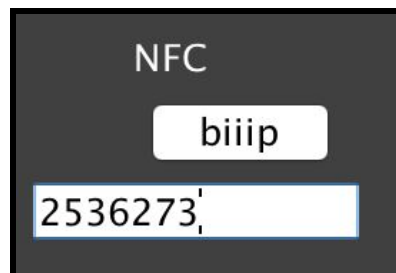
Notre machine à café permet également de gérer les stocks d'ingrédients afin de s'assurer de leur disponibilité. Ainsi, s' il manque un ingrédient à une recette ou une option,, elle sera indisponible pour les utilisateurs. Nous avons fait le choix d'afficher sur la droite de la machine un résumé des stocks d'ingrédients de celle-ci (voir ci-dessous).



Stock

- Coffee Dose = 15
- KG Of Grain = 0.74
- Tea Bag = 5
- Sugar Dose = 50
- Soup Dose = 2
- Spicy Dose = 50
- Crouton Dose = 5
- Vanilla Dose = 1
- Milk Dose = 5
- Maple Syrup Dose = 5
- Nitrogen Dose = 1
- Number of Goblet = 15

Fidélisation NFC



Le paiement NFC de notre machine offre un système de fidélisation qui permet au bout du 10ème achat, d'avoir une boisson offerte (dans la moyenne de prix des 10 achats précédents). Afin d'identifier les différentes cartes bancaires des utilisateurs, il suffit de renseigner le numéro de carte avant de payer la boisson dans le champ de texte prévu à cet effet. A noter que le nombre de commandes de chaque utilisateur est sauvegardé dans un fichier texte du projet. Ainsi le système de fidélisation est fonctionnel même après avoir relancé le programme plusieurs fois.

2) Extensions

Notre machine contient toutes les extensions demandées et nous allons voir en détail les choix d'implémentation que nous avons fait concernant celles-ci.

Gestion de la soupe

The interface for selecting soup features three sliders. On the left, text indicates the selection: 'Vous avez choisi soupe', 'Prix : 0.75€.', 'Votre solde : 0.0€.', and 'Veuillez choisir un niveau d'épice (peut être nul)'. The 'Spicy' slider is at 0, 'Size' is at Normal, and 'Temperature' is at 60°C.

| Spicy | Size | Temperature |
|-------|--------|-------------|
| 0 | Short | 20°C |
| 1 | | 35°C |
| 2 | Normal | 60°C |
| 3 | | |
| 4 | Long | 85°C |

Comme on peut le voir sur l'image à gauche, notre machine permet de commander une soupe. Ainsi, le slider de sucres se change en épices et la préparation ne peut commencer que lorsque l'utilisateur a pris conscience du slider d'épices. Cela veut dire que tant qu'il n'a pas modifié la position du slider ou placé le curseur dessus, la préparation ne se lancera, et ce même si il a inséré l'argent nécessaire.

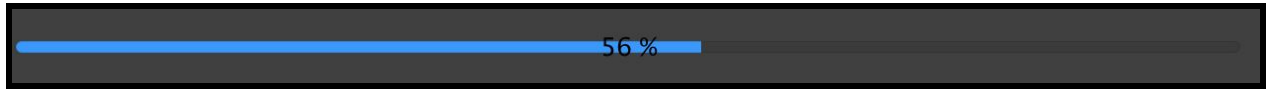
Gestion de l'iced tea

The interface for selecting iced tea features three sliders. On the left, text indicates the selection: 'Vous avez choisi thé glacé', 'Prix : -0.75€ normal size', '-1.0€ long size', and 'Votre solde : 0.0€.'. The 'Sugar' slider is at 1, 'Size' is at Normal, and 'Temperature' is at 7°C.

| Sugar | Size | Temperature |
|-------|--------|-------------|
| 0 | | 1°C |
| 1 | Normal | 4°C |
| 2 | | 7°C |
| 3 | | |
| 4 | Long | 10°C |

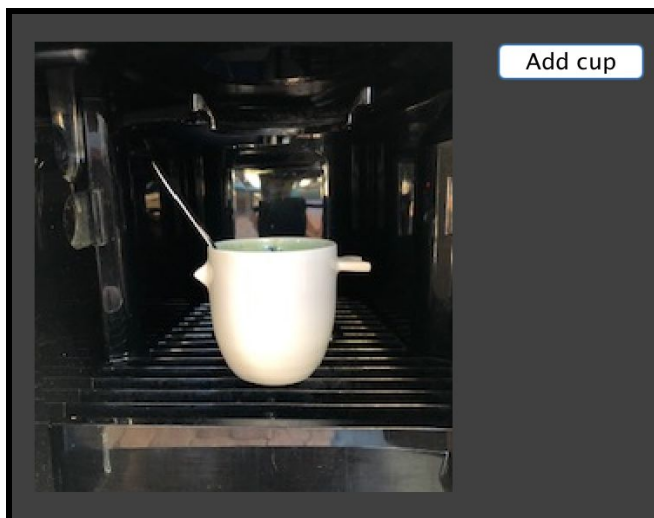
La gestion de l'iced tea est fonctionnelle et lorsqu'il est sélectionné, les sliders de taille et de température se mettent à jour. De plus, le prix n'est pas le même en fonction de la taille sélectionnée.

Gestion de l'avancement de la préparation



La progress bar est également fonctionnelle et avance pourcent par pourcent tout le long de la préparation. Ainsi, elle est plus ou moins rapide en fonction de la durée de préparation de la boisson sélectionnée et des options rajoutées.

Détection des gobelets



Pour la détection de la tasse d'un utilisateur, un simple bouton "add cup" permet d'informer la machine que celui-ci à ajouter sa propre tasse. Ainsi, le prix de la boisson est réduit de 10 centimes et lors de la préparation, la machine n'effectue pas le placement d'un gobelet, la préparation est donc plus rapide.

3) Version finale

Pour résumer globalement, nous avons implémenté toutes les exigences demandées ainsi que toutes les extensions. Voici à quoi ressemble notre machine dans sa version finale.

Coffee

Expresso

Tea

Soup

Iced Tea

Vous avez choisi café

Prix : 0.35€.

Votre solde : 0.0€.

Sugar

01234

Size

ShortNormalLong

Temperature

20°C35°C60°C85°C

Coins

0.50 €

0.20 €

0.10 €

NFC

biip

Cancel

Stock

Coffee Dose = 15

KG Of Grain = 0.74

Tea Bag = 5

Sugar Dose = 50

Soup Dose = 2

Spicy Dose = 50

Crouton Dose = 5

Vanilla Dose = 1

Milk Dose = 5

Maple Syrup Dose = 5

Nitrogen Dose = 1

Number of Goblet = 15

Options

☐ Nuage de lait (+0.10€)

☐ Sirop d'érable (+0.10€)

☐ Glace vanille mixée (+0.60€)

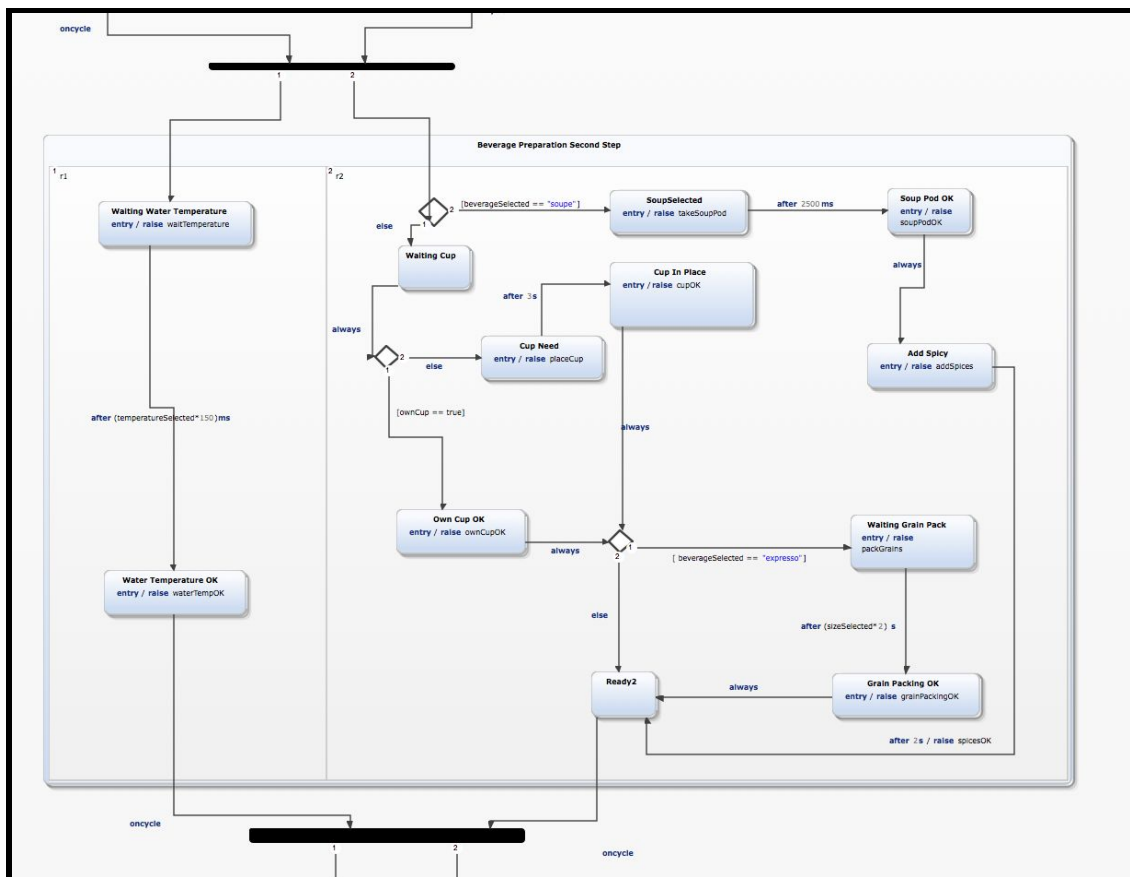


Add cup

Choix de conception

1) Conception de la machine à état fini

a) Orthogonal state



Nous avons décidé d'utiliser les *orthogonal states* Yakindu afin d'exécuter des statecharts en parallèle lors de la préparation des différentes boissons. De plus, les *synchronizations* nous ont permis d'attendre que les 2 parties de l'orthogonal state soient terminées avant d'exécuter la suite de notre machine à état (voir l'exemple ci-dessus).

b) Gestion du temps

Nous avons décidé au commencement de la machine à état de ne pas forcément réunir la gestion du temps dans notre machine à un seul endroit car cela nous semblait plus complexe à mettre en place. Cependant, cette décision nous a apporté quelques contraintes par la suite puisqu'il a fallu modifier notre machine à plusieurs endroits lorsque nous ajoutons de nouvelles actions utilisateurs (Bouton, CheckBox,...). En effet, les exigences liées aux 45 secondes d'attente avant la réinitialisation de la machine nous ont causé quelques problèmes à cause de la façon dont nous avons traité le temps.

c) Machine sur écoute

Nous avons fait le choix d'ajouter une variable permettant d'indiquer lorsque la machine est sur écoute ou non. Autrement dit lorsqu'elle peut recevoir des actions de l'utilisateur ou lorsqu'elle est bloquée. Ainsi à chaque validation de paiement et durant toute la durée de préparation d'une boisson, la machine est bloquée afin d'empêcher l'utilisateur d'effectuer des actions.

d) Séparation des systèmes de paiement

Enfin, nous avons fait le choix de séparer les systèmes de paiement du statechart principal afin de pouvoir les traiter en parallèle. Ils sont bien entendu bloqués lorsque la machine prépare une boisson.

2) Implémentation d'une interface

Pour avoir une meilleure lisibilité du code, nous avons créé une interface `Calculator` qui regroupe toutes les méthodes purement mathématiques.

Certaines méthodes dans cette interface permettent de calculer le temps qu'une boisson mettra pour se préparer. Elles sont donc plutôt complexes car elles doivent calculer le temps de plusieurs chemins simultanés et trouver le plus long pour le renvoyer.

Les autres méthodes sont des méthodes basiques qui soit déterminent la valeur moyenne d'une liste soit arrondissent un nombre décimal.

Nous avons fait le choix de mettre le type statique à toutes ces méthodes car il n'est pas vraiment utile de créer un objet juste pour les utiliser.

3) Utilisation de threads

L'utilisation de threads s'est avérée utile pour 3 choses distinctes.

Premièrement, pour contrer certains dysfonctionnement de Yakindu, nous avons décidé de faire fonctionner notre machine à état fini en *CycleBased*. Pour que la machine à état fini interagisse toujours avec le code, il nous fallait préciser la commande *runCycle* qui récupère l'état de la machine à état fini pour l'appliquer au code. Le problème est que la machine à état étant en *CycleBased*, se calcule de façon périodique. Nous avons donc utilisé un `while true` dans un *runnable* pour pouvoir appliquer la commande *runCycle* à chaque fois. Et pour ne pas que l'exécution du code se bloque au niveau du `while true`, nous avons créé un thread.

Deuxièmement, nous avons utilisé un thread pour répondre à une exigence client.

Lors de la préparation de la soupe, pour que l'utilisateur ait le temps de choisir une valeur explicite sur le slider avant la préparation de la boisson chaude, nous avons mis en place un système qui détecte la position de la souris.

Une fois que le curseur de la souris est allé sur le slider puis s'est retiré, la préparation peut commencer. Pour faire cela, nous avons utilisé deux `while true` dans un thread, l'un permet de savoir si le curseur de la souris est arrivé sur le slider et si oui, déclenche le second `while true`. Donc le second `while true` qui est englobé par le premier `while true` permet de savoir si le curseur de la souris a quitté le slider.

Nous ne pouvions pas répondre à cette exigence client avec le thread de la machine à état fini puisque si nous cassons la boucle de ce thread, les états de la machine à état fini ne sont plus récupérés par le code. Et nous devons impérativement casser la boucle car elle permet aussi de délayer le paiement (attendre l'action utilisateur pour préparer la soupe s'il a payé avant).

Aussi, mettre un `while true` dans le `while true` du thread de la machine à état fini bloquera totalement le thread et fera planter le programme qui ne pourra pas avancer car aucune information ne sera récupérée en provenance de la machine à état finis.

Enfin, nous avons utilisé un thread pour implémenter la barre de progression. Pour ajouter des pourcentages à la barre de progression il suffit de mettre un `while true`, et à chaque passage dans la boucle, on ajoute 1%.

Le problème ici est d'ajouter les pourcentages à la barre de progression tous les centièmes du temps de la préparation. Nous avons donc deux méthodes à sonder, utiliser un nouveau thread avec la méthode `sleep` ou alors implémenter les timers de java dans le thread de la machine à état fini qui permettent d'exécuter le code après un certain temps.

La méthode `sleep` des threads n'est pas forcément optimale car la précision des sleeps dans les threads est limitée par les fonctionnalités fournies par le système d'exploitation sous-jacent. Ajouter à cela le fait que le thread de la machine à état fini sommeille une durée toujours fixe, indépendante du temps de préparation, il nous était pas possible d'utiliser les timers dans le thread de la machine à état tout en ayant une barre de progression précise.

C'est pourquoi nous avons créé un nouveau thread, celui de la barre de progression. Nous le faisons donc sommeiller tous les centièmes du temps de la préparation de la boisson.

Malgré cette solution, la barre de progression peut certaines fois ne pas être très précise à cause de la nature des sleeps dans les threads. Nous ajoutons donc 1 milliseconde lorsque la barre de progression n'est pas précise à chaque fois que le thread sommeille et le problème est réglé en apparence. Car même si cela n'est pas détectable à l'œil nu, la barre de progression ne se synchronise pas parfaitement mais cela n'est pas impactant pour une interaction avec le client.

4) Choix des classes

Pour implémenter les 5 types de boissons, nous avons créé un enum Beverage dans lequel nous avons renseigné les informations des boissons. Nous avons hésité à utiliser le concepte d'héritage en java pour créer une classe abstraite beverage et créer des subclass qui représentent chacune un type de boisson, mais cela ne permettait pas une bonne extensibilité du code car plus il y a de boisson, plus il y a de classes. C'est pourquoi nous avons utilisé un *enum* à la place. Ici l'enum convient bien pour représenter un petit ensemble de données que sont les boissons.

De plus, nous avons également créé une classe supply. Cette classe sert à implémenter les ingrédients et permet de savoir s'il y a les ingrédients nécessaires pour la préparation d'une boisson et/ou des options. Les variables de cette classe sont initialisées par défaut par le constructeur.

Enfin, nous avons créé une classe *WriteAndDecodeFile*. Cette classe permet de créer un fichier texte, d'y accéder en y modifiant et en y récupérant ses données. Nous avons créé cette classe pour permettre une meilleure lisibilité du code. La classe étant *DrinkFactoryMachine* très grande principalement à cause des méthodes qui l'initialise avec son interface, la priorité était de ne pas surcharger la classe.

Le but de cette classe est d'octroyer la possibilité de stocker les informations des utilisateurs par carte bleue même si l'application se ferme. La classe *WriteAndDecodeFile* permet de stocker les informations des utilisateurs qui ont payé une boisson par carte bleue. Pour stocker les données, nous avons utilisé une Map plus particulièrement une HashMap. Pour chaque utilisateur qui paie par carte bleue, une clé associée à une liste de valeur est rajoutée dans la HashMap si la clé n'existait pas avant. Si la clé existait alors la valeur est ajoutée à la liste de valeur correspondant à la

bonne clé. Nous avons décidé d'utiliser une HashMap pour être sûr qu'il n'y ait pas de possibilités que des doublons se forment pour faciliter la lecture de données.

En effet, s'il y a beaucoup d'utilisateurs, il nous faut un moyen rapide de récupérer les informations, c'est pourquoi la HashMap s'est avérée plus juste qu'une simple liste.

Mais comme le concept de Map n'admet pas de doublons, alors il nous a fallu une liste pour stocker les valeurs.

Activité de vérification et validation par ltsa

1) Propriétés LTL

Nous avons à travers ltsa, représenté plusieurs sous modèles de notre machine à état. Pour chaque sous-modèle que nous avons représenté, nous avons défini des propriétés LTL. Le but était de demander au système si les propriétés que nous définissons et qui doivent être respectées le sont effectivement. Nous avons donc créé 10 propriétés pour tous les sous modèles de notre machine à état sur ltsa. Les priorités nous ont permis de définir un processus déterministe qui affirme que toutes traces comprenant des actions dans un alphabet X, est acceptée par X. Cela a montré que les sous-modèles de notre machine à état satisfont les propriétés requises de son comportement.

2) Sécurité

La propriété *security* de ltsa est sans aucun doute la plus importante. C'est cette propriété qui nous informe si quelque chose d'imprevu peut arriver dans notre machine à état.

Quand nous demandons à ltsa de vérifier la sécurité des machines à état que nous avons implémentés, nous ne trouvons pas de faille.

Il y a tout de même un potentiel *deadlock* qui, nous le pensons, est là à juste titre. Comme une de nos machines à état finis doit implémenter une synchronisation, c'est-à-dire qu'il doit attendre qu'une transition se fasse dans un *process*, un *deadlock* se forme si l'un uniquement l'un des deux *process* et finis et l'autre non. Mais cela est normal car c'est le but même d'une synchronisation.

3) Vérifications additionnelles

Nous avons aussi utilisé *supertrace* qui nous aidé à voir le si le potentiel *deadlock* d'un de nos sous-modèles était vraiment critique et constituait une mauvaise implémentation. Mais la *supertrace* n'a rien trouvé de choses qui aurait pu nous indiquer une mauvaise implémentation.

Enfin, nous avons regardé si toutes nos machines à états étaient bien dessinables en s'assurant qu'elles n'ont pas une taille trop disproportionnée pour ce qu'elles doivent faire. Nous avons aussi consulté l'alphabet de chacune.

Prise de recul

Pour conclure, globalement, nous sommes plutôt satisfaits du travail que nous avons réalisé au cours de ce projet. En effet, nous nous sommes grandement investis dans celui-ci afin d'accomplir toutes les tâches demandées. Cependant, il est tout à fait possible que notre machine présente quelques défauts et petits bugs que nous n'aurions peut être pas observés lors de nos tests et donc pu corriger.

Au niveau de la conception de notre machine à état, il y a quelques points pour lesquels nous aurions fait sans doute différemment.

Le premier point est la détection des actions utilisateurs avant le lancement de la préparation d'une boisson. En effet, nous n'avons pas réuni la gestion du temps à un seul endroit dans notre machine à état ce qui a entraîné différentes contraintes au cours de l'implémentation des fonctionnalités.

Ensuite, nous aurions conçu une partie de notre statechart différemment pour utiliser la méthode *Event_driven* au lieu de *Cycle_base* qui nous paraît légèrement brutal.

Enfin, nous aurions aussi évité de mélanger les langages dans notre projet. En effet, nous avons mélangé l'anglais et le français dans notre interface graphique ce qui peut s'avérer contradictoire.