Enseirb-Matmeca, Filière Informatique

Conception de logiciel - Méthode B

Rapport de Projet

Auteurs :
Nada Ben Abdelkrim
Benjamin Briand
Florian Chouarain
Anas Hakkal

Encadrant :
Pierre Castéran

Table des matières

Introduction

Ce projet intervient dans le cadre de l'enseignement "Conception de logiciels", un module qui nous a fait découvrir la méthode B utilisée dans Rodin et qui permet le développement d'un logiciel prouvé correct. En effet elle est souvent utilisée pour les logiciels critiques ; le client demande des spécifications semi-formelles, pour éviter toute mauvaise compréhension par le concepteur, les mathématiques sont utilisées comme un langage de communication entre les deux. L'objectif de ce projet est de formaliser une situation correspondant à un arrêt de bus : le bus arrive à un arrêt, des passagers descendent, d'autres y montent, puis le bus repart. Il s'agit de traiter le fonctionnement d'un seul arrêt.

Pour toutes nos machines, l'initialisation représentera donc l'arrivée du bus à l'arrêt.

Nous disposons de plusieurs contraintes à respecter, que l'on décrira au cours de ce rapport (en gras), au fur et à mesure qu'elles seront traitées. Une première machine devra respecter certaines de ces contraintes, puis des raffinements nous permettront de formaliser la situation initiale de manière plus complète, c'est-à-dire en respectant de plus en plus de contraintes.

La dernière machine, obtenue à partir de la première machine, et de l'ensemble de ses raffinements successifs, devra respecter toutes les contraintes, et comporter une condition de non-blocage.

1 Une première machine

La première machine est d'une grande importance. C'est la seule façon pour vérifier que c'est le modèle demandé par le client, elle doit être simple et abstraite pour faciliter sa compréhension. Notre première machine se nomme Bus. Elle est liée au contexte "Users", qui comprend un ensemble Users d'une taille finie, contenant les passagers du bus ainsi que les personnes à l'extérieur du bus (c'est-à-dire les personnes qui attendent à l'arrêt, mais également des personnes pouvant arriver à l'arrêt).

Ce contexte définit également une capacité $n \in \mathbb{N}$, correspondant au nombre maximal de passagers.

Nous avons défini 4 ensembles (qui appartiennent à Users) :

— passengers, qui contient les passagers du bus. On utilise un invariant pour s'assurer que cet ensemble a une taille inférieure ou égale à n. Cela nous permet de respeter la deuxième contrainte du projet : **"Le bus a une capacité maximum en nombre de voyageurs, capacité qui ne peut être dépassée"**.

— waiting, qui contient les passagers qui attendent à l'arrêt.

— leaving_passengers (inclu dans passengers) : il s'agit des passagers voulant quitter le bus.
— gone_passengers : ils s'agit des passagers qui ont quitté le bus. Nous choisissons ainsi de ne pas faire rentrer de nouveau dans le bus les personnes qui viennent d'en descendre.

À partir de cela, nous avons défini plusieurs évènements.

Tout d'abord, à l'initialisation, nous nous assurons que les passagers appartiennent à notre ensemble Users, qu'il y a au plus n passagers, et que ceux voulant quitter le bus appartiennent bien aux passagers du bus. Nous vérifions également que personne n'a quitté le bus et qu'il n'y a aucune personne qui attend à l'arrêt : cela nous permet de respecter la contrainte : **"On supposera qu'à l'initialisation, aucun usager n'attend à l'arrêt. En revanche le bus contient un nombre quelconque de passagers initiaux (compatible avec la capacité de ce véhicule)."**

Nous avons ensuite défini les évènements liés à la première contrainte : **"Dans les machines les plus concrètes, la montée ou la descente d'un voyageur doit être un événement élémentaire."**

L'évènement leave_one a pour but de faire sortir un passager du bus. Nous avons choisi de faire sortir les passagers du bus un par un pour deux raisons :
— La première est liée à des questions de réalisme : dans la vie réelle, les passagers ne sortent pas du bus en même temps.
— Aussi, l'un des objectifs est de pouvoir faire entrer à tout moment des personnes dans la file d'attente de l'arrêt de bus, notamment lorsque des passagers descendront du bus. On n'aurait pas pu faire cela si tous les passagers descendaient d'un coup.

Il était donc préférable de les faire sortir un par un, plutôt que de faire sortir l'ensemble des passagers d'un coup.

Cet évènement sera réalisé tant que des passagers voudront descendre (c'est-à-dire tant que leaving_passenger n'est pas un ensemble vide). Les passagers ayant quitté le bus sont ajoutés à l'ensemble gone_passengers.

Dans le même temps, nous pouvons faire entrer les passagers qui attendent à l'arrêt.

Là, nous avons été confronté à une petite difficulté. Nous ne pouvions pas faire rentrer tous les passagers du bus en même temps : en effet, le bus ayant une capacité limitée, si les passagers du bus, plus ceux de la file d'attente étaient plus nombreux que la capacité, nous n'aurions pu faire entrer aucun passager. De plus, il n'est pas très réaliste de faire entrer tous les passagers en même temps.

C'est pour ces raisons que nous avons défini l'évènement enter_one, qui peut être réalisé lorsque le bus n'est pas plein, et qui fait entrer un passager de l'ensemble waiting dans le bus.

Enfin, l'évènement enter_one_waiting permet de faire entrer une personne dans la file d'attente (c'est-à-dire dans waiting). Il peut être effectué à tout moment si il y a des passagers qui ne sont ni dans le bus, ni dans la file d'attente.

Ainsi, notre première machine respecte trois contraintes (mentionnées en gras ci-dessus). Nous traiterons les autres contraintes dans des raffinements, notamment parce que certaines sont plus difficiles à mettre en oeuvre.

2 Un premier raffinement

Après avoir réalisé une première machine traitant certaines des contraintes les moins difficiles à mettre en oeuvre, nous avons réalisé une deuxième machine, Bus1, qui raffine notre machine Bus. Cela signifie que tous les comportements de Bus1 correspondent à ceux de Bus, mais également que Bus1 est plus complète que Bus, et traite plus de contraintes.

Tout d'abord, nous avons défini le contexte Users1 qui étend le contexte Users, dans lequel on définit une fonction priority qui attribue à un utilisateur un nombre naturel correspondant à sa priorité pour monter dans le bus.

Pour cette nouvelle machine, nous faisons entrer les passagers en fonction de leur priorité : pour chaque montée dans le bus, le passager entrant sera l'un de ceux ayant la plus grande priorité. La nouvelle contrainte que nous voulons respecter est la suivante : **"Chaque usager possède un niveau de priorité pour monter dans le bus (0 pour les voyageurs lambda, strictement positif pour les catégories usuelles)"**. Les évènements de Bus1 raffinent ceux de Bus, et l'évènement à compléter dans Bus1 (par rapport à Bus) est celui permettant l'entrée dans le bus : enter_one. Il faut rajouter une garde afin de signaler que la personne qui entre dans le véhicule a une priorité supérieure à toutes les personnes qui attendent à l'arrêt.

Aussi, les passagers font maintenant preuve de civisme : pour l'évènement lié à l'entrée des passagers dans le bus, nous avons rajouté une garde (aussi dans enter_one) vérifiant que leaving_passenger est vide, afin de vérifier que tous les passagers voulant sortir sont bien hors du bus. Nous respectons donc la contrainte : **"Par civisme, les usagers ne montent dans le bus que lorsque les voyageurs voulant en descendre l'ont fait"**.
Nous respectons donc deux contraintes supplémentaires.

3 Un deuxième raffinement

Afin de traiter la dernière contrainte, nous avons effectué une nouvelle machine, Bus2, qui raffine Bus1 (qui est notre premier raffinement). Cette machine doit aussi s'occuper des propriétés de non-blocage.

La contrainte est la suivante : **"À priorité égale, c'est l'ancienneté dans la file d'attente qui est déterminante"**. Il va donc falloir garder en mémoire l'ordre d'arrivée de chaque utilisateur à l'arrêt de bus (les utilisateurs qui attendent d'entrer dans le bus sont dans l'ensemble waiting). Nous avions initialement pensé à réaliser une file pour stocker les utilisateurs attendant à l'arrêt de bus. Cependant, cela n'était pas possible, car l'utilisateur à faire entrer dans le bus ne sera pas forcément celui qui attend depuis le plus longtemps, le premier critère à prendre en compte étant la priorité.

Nous avons choisi de définir une fonction f qui attribue à chaque utilisateur arrivant à l'arrêt de bus son ordre d'arrivée. Elle vaudra 0 pour le premier arrivé à l'arrêt, puis sera incrémentée de 1 pour chaque nouvel arrivant. Cela sera réalisé dans l'évènement enter_one_waiting, notamment à l'aide d'un compteur initialisé à 0. Dans l'évènement enter_one, on rajoute une garde vérifiant que si plusieurs utilisateurs ont la même priorité, on choisit bien celui étant arrivé le premier.

Propriétés de non-blocage

Le non blocage est une propriété essentielle : elle caractérise la capacité du système à exercer une activité au cours de sa durée de vie, c'est-à-dire le fait qu'une action est toujours possible, tant que la situation formalisée n'est pas terminée. Ici, il s'agit de vérifier qu'il est toujours possible d'effectuer un évènement tant que l'on n'est pas arrivé dans l'évènement bus_leaving, c'est-à-dire tant que le bus n'a pas quitté la station.

Il s'agit en fait de vérifier, à l'aide d'invariants que tant que l'on n'est pas entré dans l'évènement bus_leaving, toutes les gardes d'au moins un évènement peuvent être traitées sont vérifiées.

Nous créons donc l'invariant DLF dans lequel nous traitons quasiment tout ce qui nous assure le non-blocage.

Nous créons un autre invariant vérifiant que si il y a des personnes à l'arrêt de bus, alors il y a, parmi les utilisateurs ayant la priorité maximale, un premier arrivé.

Nous avons mis cet invariant à part car il est très difficile à vérifier. En effet il fait appel à des propriétés mathématiques que Rodin a du mal à assimiler. Nous avons donc dû l'aider un petit peu en insérant dans le contexte un axiome indiquant que, parmi les utilisateurs d'un sous-ensemble quelconque de Users, il existe un ou plusieurs utilisateurs ayant une priorité maximale, c'est-à-dire que la fonction priorité a un maximum.

Nous avons également dû ajouter plusieurs invariants afin de traiter cela, notamment l'invariant 10 indiquant que parmi n'importe quel sous-ensemble des utilisateurs attendant à l'arrêt (celui qui nous intéresse est le sous-ensemble des utilisateurs de priorité maximale qui attendent à l'arrêt), il y en a un qui est arrivé en premier, c'est-à-dire que la fonction f a un minimum, pour les utilisateurs qui attendent à l'arrêt.

Cette fonction est à 0 pour le premier utilisateur, et est incrémentée de 1 pour chaque utilisateur. L'invariant 8 vérifié par Rodin nous permet de nous assurer que la fonction f renvoie un résultat différent pour des utilisateurs différents en entrée. De plus, dom(f) est un ensemble fini (car inclu dans l'ensemble Users qui est fini, comme indiqué dans le contexte Users), donc cet invariant nous apparait comme correct.

Cependant, cet invariant n'est pas vérifié par Rodin qui a des difficultés à prouver tout ce qui est lié à des propriétés mathématiques faisant appel à des notions de maximum et de minimum.

Convergence des évènements

Il s'agit ici de vérifier que les évènements convergent bien vers l'évènement bus_leaving, c'est-à-dire que le bus va pouvoir quitter l'arrêt.

Juste après l'initialisation, deux évènements peuvent être effectués : leave_one (la sortie des passagers du bus) et enter_one_waiting (l'arrivée d'utilisateurs dans la file d'attente).

L'évènement leave_one ne peut s'exécuter que si l'ensemble des passagers voulant quitter le bus (leaving_passenger) n'est pas vide. Or, cet ensemble est fini et perd un élément dans l'évènement leave_one. Il ne peut récupérer des éléments nulle part après l'initialisation. On ne peut donc entrer dans cet évènement qu'un nombre fini de fois.

Aussi, l'évènement enter_one_waiting ne peut s'exécuter que si l'ensemble des utilisateurs qui n'ont jamais été dans le bus et qui ne sont pas à l'arrêt n'est pas vide (cet évènement consiste à les faire arriver à l'arrêt de bus). Or, cet ensemble est fini puisque l'ensemble des utilisateurs est fini. Après l'initialisation, aucun élément n'est ajouté dans cet ensemble dans tous les évènements qui composent notre machine. Par contre, on enlève un élément de cet ensemble à chaque fois que l'on entre dans l'évènement enter_one_waiting. On ne peut entrer dans cet évènement qu'un nombre fini de fois.

Lorsque tous les passagers sont sortis (c'est-à-dire que l'ensemble leaving_passenger est vide), on peut exécuter l'évènement enter_one, qui permet l'entrée d'un passager dans le bus. Cet évènement ne peut s'exécuter que tant que le bus n'est pas plein. Or, lorsque l'on peut entrer dans cet évènement, on ne fait plus sortir de passagers

du bus. Comme on fait entrer des passagers dans le bus lors de cet évènement et que le bus a une capacité finie, on ne peut exécuter cet évènement qu'un nombre fini de fois.

De plus, on ne peut effectuer cet évènement que si il y a des passagers à l'arrêt. Des passagers peuvent arriver à l'arrêt. Mais l'ensemble des personnes pouvant arriver à l'arrêt et étant à l'arrêt est fini. Et comme cet évènement retire une personne de l'arrêt, il ne peut être exécuté qu'un nombre fini de fois également.

Ainsi, les trois évènements (leave_one, enter_one et enter_one_waiting) suivant l'initialisation vont être effectués un nombre fini de fois : on va donc forcément converger vers l'évènement bus_leaving, qui correspond au départ du bus de l'arrêt.

4 Difficultés rencontrées

Ce projet a été une nouveauté dans notre formation, il s'agit d'une nouvelle manière de concevoir un logiciel : une perspective plus théorique et abstraite par rapport aux autres projets qu'on a réalisé. Même si le cours a été complet et détaillé, nous nous sommes confrontés à quelques difficultés durant la réalisation de ce projet :

- Les messages d'erreur ne sont pas clairs : nous avons trouvé une difficulté à comprendre les messages d'erreur ou des obligations de preuves. Les messages ne sont pas toujours parlants et la ligne indiquée ne correspond pas toujours à la source de l'erreur.
- Rodin est un outil peu utilisé, en comparaison des autres outils utilisés à l'ENSEIRB nous n'avons trouvé que peu de documentation sur internet.
- Nous ne sommes pas arrivés à prouver toutes les obligations de preuves (cf paragraphe Propriétés de non-blocage).

Conclusion

Ce projet nous a permis d'apprendre la programmation en langage B et de comprendre son fonctionnement. Si nous avions déjà pu la manipuler au cours des TDs, ce projet nous a permis de l'utiliser pour un travail plus long et complet, demandant plus d'autonomie.

Aussi, nous avons pu explorer plus en profondeur les fonctionalités de Rodin qui est un outil auquel nous ne sommes pas habitués, notamment les prouveurs et l'arbre des preuves, afin de valider certaines obligations de preuve plus complexes.

De plus, le fait de travailler en groupe est très enrichissant et permet un partage des connaissances qui est très bénéfique pour notre formation.

En effet ce projet était une opportunité de découvrir qu'avant même de réaliser un logiciel, on peut prouver qu'il est correct c'est-à-dire qu'il répond aux spécifications

et satisfera le client. Alors que les méthodes basées sur des tests permettent juste d'affirmer que les testeurs n'ont pas trouvé d'écart entre la spécification et le code exécuté. Nous avons aussi appris à corriger des bugs à l'aide de Rodin.

Ce module nous a permis de découvrir des choses assez différentes de ce que l'on a l'habitude de faire en informatique.