TP8, 9 et 10 - Vérification d'un système de négociation tarifaire

Fichiers du projet : `tp89.thy`, `TP89_ACF.zip` et fichiers d'aide : `table.thy`, `tp89proof.thy`, `pc.thy`

Le TP est à rendre **sur le MOODLE du cours**. Déposez sur MOODLE un archive au format **ZIP** contenant :

1. **un JAR** de votre projet `TP89_ACF` au format `Nom1_Nom2_TP89.jar`. Pour générer le `.jar`, dans le shell SBT, tapez `package`. Le `.jar` doit être généré dans le répertoire `target/scala-2.13` du projet. **Renommez** le `.jar` avec le nom de votre binôme `Nom1_Nom2_TP89.jar`. Remarque : le `.jar` n'a pas besoin d'être exécutable.
2. **la théorie Isabelle** : `tp89.thy` (et éventuellement `table.thy` si vous l'utilisez).
3. Un fichier au format .zip de **votre répertoire de projet TP89_ACF complet**.
   Tout votre code Scala devra être contenu dans le fichier `ConcreteValidator.scala`, voir remarque **IMPORTANT** sur la page suivante.

On se propose de faire un agent de validation de négociations commerciales. Cet agent, déployé par une banque, reçoit des demandes de transactions émanant de clients et de marchands. Le rôle de cet agent est de construire la liste des transactions validées. Une transaction est validée lorsque le montant `amc` proposé par le client est supérieur ou égal au montant `amm` demandé par le marchand. Dans ce cas, le montant retenu pour la transaction est `amc`. Toute transaction validée ne peut être renégociée : si une transaction a été validée avec un montant `am` celui-ci ne peut être changé par la suite. Chaque transaction est identifiée par un triplet `(c,m,i)`, avec `c,m,i` entiers naturels. L'entier `c` identifie un client, `m` identifie un marchand et `i` est un numéro de transaction entre `c` et `m`. La liste de transactions validées est une liste de couples `((c,m,i),am)` où `(c,m,i)` est le triplet identifiant la transaction et `am` l'entier naturel représentant le montant retenu pour la transaction.

Les clients peuvent envoyer à l'agent de validation des messages de la forme `(Pay (c,m,i) am)` et les marchands peuvent envoyer des messages de la forme `(Ack (c,m,i) am)` ou `(Cancel (c,m,i))` où `am` est un naturel qui peut être égal à 0. En revanche, il n'est pas possible de valider une transaction avec un prix nul. A noter que l'ordre de réception des messages est quelconque, en particulier ce n'est pas nécessairement au client de commencer. Dans ces messages, les montants `am` sont des entiers naturels représentant les montants proposés pour la transaction. Un message de la forme `(Pay (c,m,i) am)` signifie que le client `c` accepte de payer à `m` le montant `am` pour la transaction `i`. Un message de la forme `(Ack (c,m,i) am)` signifie que le marchand `m` demande au client `c` un montant `am` pour la transaction `i`. Un message `(Cancel (c,m,i))` signifie que le marchand `m` annule toute transaction `i` réalisée avec `c`. A noter que si le message `(Cancel (c,m,i))` est reçu avant tout `Pay` ou `Ack` sur `(c,m,i)`, celui-ci empêche la réalisation de cette transaction.

Si, pour une transaction, un client a proposé un montant `am`, tout montant `am'` inférieur proposé par la suite est ignoré par l'agent de validation (ce message ne fait pas progresser la négociation). De la même façon, si un marchand a proposé un montant `am`, tout montant `am'` supérieur proposé par la suite est ignoré par l'agent de validation. A l'inverse et si la transaction n'est pas encore validée, tout montant supérieur proposé par le client et tout montant inférieur proposé par le marchand est pris en compte par l'agent.

Préambule

1. Copiez et chargez dans Isabelle le fichier `tp89.thy`. Ce fichier contient les définitions : `transid` le type des identifiants de transaction, `transaction` le type des transactions validées et `message` le type abstrait des messages.
2. Décompressez l'archive `TP89_ACF.zip` dans votre répertoire ACF. Dans votre éditeur Scala, ouvrez le projet `TP89_ACF`. Dans le projet `TP89_ACF`, vous devez remplacer la chaîne de caractères `"<LE_NOM_DE_VOTRE_BINOME>"`.

Pour lancer le projet, exécuter la classe Java `bank.Ihm`. La classe responsable de la validation des transaction est la classe `validator."<LE_NOM_DE_VOTRE_BINOME>".ConcreteValidator`. Cette classe contient les **deux** méthodes à implémenter : `process(e : message)` qui traite un message et la méthode `getValidTrans` qui retourne une liste de transactions validées.

IMPORTANT : Tous les objets et classes nécessaires à l'exécution de votre solution seront à ajouter dans le fichier `ConcreteValidator.scala` (et **uniquement** celui-ci), mais **en dehors** de la classe `ConcreteValidator`. Actuellement, `ConcreteValidator.scala` contient un objet `tp89` qui permet de pouvoir compiler le projet tel quel. Il faut le remplacer par le code généré par Isabelle/HOL. A noter qu'un certain nombre d'objets et classes générés à partir de la théorie Isabelle/HOL sont déjà présents dans d'autres fichiers et ne devront pas être recopiés dans `ConcreteValidator.scala`. C'est le cas de l'objet `Nat`, de la classe `Nat`, de l'objet `Natural` et de la classe `Natural` qui sont déjà présents dans le fichier `Bank.scala` qui est fourni.

1 Evaluation

L'évaluation de ce TP se passera en deux temps :

1. D'abord, vous allez développer l'outil de validation des transactions en Isabelle/HOL, exporter le code Scala correspondant et vérifier son intégration avec l'interface graphique fournie. Le premier document à rendre sera le code de cet outil.
2. Dans un deuxième temps, tous les outils de tous les binômes seront mis en ligne sur internet. Vous pourrez attaquer tous les TPs de vos camarades sur un serveur dédié. Par attaque, on entend une séquence de messages envoyés qui provoque un comportement qui ne respecte pas la spécification donnée plus haut et détaillée sous forme de propriétés dans la section 5. Les attaques seront enregistrées automatiquement par le serveur, vous n'aurez qu'à cliquer sur le ou les numéros de propriétés violées par l'attaque.

2 Marche à suivre

1. Réaliser en Isabelle/HOL l'outil de validation des transactions qui à partir d'une séquence de messages construira la liste des transactions validées.
2. Ecrire les lemmes garantissant la sûreté de vos fonctions (voir Section 5).
3. Recherchez des contre-exemples à ces propriétés. **Notez ces contre-exemples, ils pourront vous servir pour attaquer, ensuite, les TPs de vos camarades !**
4. Bonus : réalisez les preuves.
5. Une fois satisfait de votre code, dans la théorie Isabelle/HOL, exportez en Scala.

6. Intégrez dans le projet TP89_ACF en complétant le fichier `ConcreteValidator.scala` **et uniquement celui-ci**. En particulier, complétez les méthodes `process` et `getValidTrans` de la classe `ConcreteValidator`. Testez et envoyez votre TP (voir consignes au début du sujet).
7. Dès qu'ils sont disponibles attaquez les TPs de vos camarades.

3 Principe de développement : "program and proof co-design"

Pour réussir à produire un outil de validation qui résistera aux attaques de vos camarades, il est conseillé de développer en parallèle les fonctions, les lemmes et les preuves ("program and proof co-design") :

1. Commencez par écrire en Isabelle/HOL les propriétés 1 à 9 attendues sur votre fonction (voir Section 5). C'est possible sur le validateur et cela vous permettra de lever certaines ambiguités.
2. Ecrivez le code Isabelle/HOL des fonctions `traiterMessage`, `traiterMessageList` et `export` (voir Section 4).
3. **Vérifiez que ces fonctions peuvent être exportées en Scala** en vérifiant que la directive `export_code traiterMessage export in Scala` génère bien du code Scala. En particulier, une fonction définie par pattern-matching sur les `nat` (cas `0` et `Suc x` par exemple) ne pourra pas être exportée. A la place vous pouvez utiliser les opérateurs d'égalité et de comparaison sur les naturels.
4. Recherchez des contre-exemples aux lemmes correspondant aux propriétés de 1 à 9.
5. **Bonus** : faites les preuves. Si vous souhaitez réaliser ce bonus, le fichier `tp89proof.thy` donne une quarantaine de lemmes intermédiaires (en français) utiles pour mener à bien la preuve complète. Le fichier `pc.thy` donne des exemples de preuves Isabelle/HOL étendant les principes vus en cours.

4 Les fonctions à réaliser

Tout d'abord, il est conseillé de construire une structure de donnée (nous appellerons son type `transBdd`) dans laquelle vous mémoriserez toute information que vous jugerez utile concernant les transactions en cours. Pour réaliser `transBdd` vous pouvez utiliser les tables d'association définie dans `table.thy`.

1. Pour le traitement des messages vous devrez réaliser une fonction `traiterMessage` qui à partir d'un message et d'un état de `transBdd` calcule un nouvel état de `transBdd` tenant compte du message reçu : `traiterMessage` :: `message` ⇒ `transBdd` ⇒ `transBdd`
2. Pour construire la liste des transactions validées à partir d'une base de données de toutes les transactions en cours, vous devrez réaliser une fonction : `export` :: `transBdd` ⇒ `transaction list`
3. Enfin, pour faciliter l'écriture des propriétés, il est conseillé d'écrire la fonction `traiterMessageList` qui traite une liste de messages et produit la base de données correspondant au traitement de tous les messages de la liste : `traiterMessageList` :: `message list` ⇒ `transBdd`

5 Propriétés attendues

A l'aide de lemmes, on s'assurera que la fonction de validation a les 9 propriétés suivantes. Définissez ces lemmes uniquement à l'aide des fonctions **visibles à l'exterieur** de l'application : `export`, `traiterMessage`, `traiterMessageList` et des opérations (prouvées) d'Isabelle/HOL, comme par exemple : `List.member`, `@` (ou `append`), `#`, ... sur les listes.

1. Toutes les transactions validées ont un montant strictement supérieur à 0.
2. Dans la liste de transactions validées, tout triplet `(c,m,i)` (où `c` est un numéro de client, `m` est un numéro de marchand et `i` un numéro de transaction) n'apparaît qu'une seule fois.
3. Toute transaction (même validée) peut être annulée.
4. Toute transaction annulée l'est définitivement : un message `(Cancel (c,m,i))` rend impossible la validation d'une transaction de numéro `i` entre un marchand `m` et un client `c`.
5. Si un message `Pay` et un message `Ack` avec un même identifiant `(c, m, i)` ont été envoyés, tels que le montant proposé par le `Pay` est strictement supérieur à 0, et est supérieur ou égal au montant proposé par le message `Ack`, et s'il n'y a pas eu d'annulation pour `(c, m, i)`, alors une transaction pour `(c, m, i)` figure dans la liste des transactions validées.
6. Toute transaction figurant dans la liste des transactions validées l'a été par un message `Pay` et un message `Ack` tels que le montant proposé par le `Pay` est supérieur ou égal au montant proposé par le message `Ack`.
7. Si un client (resp. marchand) a proposé un montant `am` pour une transaction, tout montant `am'` inférieur (resp. supérieur) proposé par la suite est ignoré par l'agent de validation.
8. Toute transaction validée ne peut être renégociée : si une transaction a été validée avec un montant `am` celui-ci ne peut être changé.
9. Le montant associé à une transaction validée correspond à un prix proposé par le client pour cette transaction.