Conception Formelle

2022-2023

TP1: Découverte de Frama-C

Vincent Penelle

Points abordés

- Prise en main de l'interface de Frama-C.
- Spécifications et preuves de fonctions simples.

Dans le cadre de ce TP (et des suivants), nous utiliserons la version 27.1 de Frama-c, avec la version 2.5.2 d'alt-ergo. C'est celle qui est installée au CREMI et celle sur laquelle les présents TP ont été testés. Par ailleurs le solveur Z3 en version 4.12.2 est également installée, ce qui vous permettra de l'utiliser également parallèlement à alt-ergo. Toutes ces versions sont celles de l'été 2023 (date de mise à jour d'opam au CREMI).

Pour travailler au CREMI, connectez-y vous grâce à x2go, comme expliqué ici[1]. Ensuite, comme frama-c y a été installé via opam, il n'est pas disponible de suite. Pour l'utiliser, vous devez suivre les points suivants: dans un terminal, tapez 'export OPAMROOT=/opt/local/opam' puis 'eval $(opam env)'. Vous aurez à faire ces deux étapes à chaque fois que vous lancerez un nouveau terminal. Une fois cela fait, frama-c sera accessible via la commande 'frama-c-gui'. Vous pouvez bien évidemment inclure ces deux commandes dans votre .bashrc pour ne pas avoir à les taper quand vous lancez un terminal, ou faire un script. Ces informations sont rappelées sur la page du cours.

Vous aurez une autre manipulation à faire (mais une seule fois) pour que frama-c reconnaisse les prouveurs : 'why3 config detect' (après avoir tapé les commandes précédentes).

Récupérer l'archive suivante[2] et décompressez-la.

Si vous souhaitez travailler chez vous, vous devrez installer frama-c. Il est vivement conseillé de le faire via opam plutôt que par le gestionnaire de paquets de votre système. La marche à suivre est détaillée pour chaque système d'exploitation ici[3]. N'hésitez pas à consulter les instructions détaillées en cas de problème. Par contre attention, cela installera les versions les plus récentes, donc il est possible que si vous faites cela à distance de la rédaction de ce sujet, vous n'ayez pas les même versions qu'au CREMI. Pour avoir les mêmes, vous pouvez, dans la dernière ligne (celle qui installe frama-c) remplacer 'frama-c' par 'z3.4.12.2 alt-ergo.2.5.2 frama-c.27.1'. Cela installera les mêmes versions qu'au CREMI et devrait fonctionner sans problème.

Dans tous les cas, même si vous travaillez chez vous, les pas listés pour travailler au CREMI (excepté la commande export qui n'est utile qu'au CREMI) restent nécessaires.

Exercice 1: Introduction à Frama-C

[1] `https://services.emi.u-bordeaux.fr/intranet/spip.php?article125`

[2] `https://vpenelle.pages.emi.u-bordeaux.fr/conception-formelle/tp1.tar.gz`

[3] `https://www.frama-c.com/html/get-frama-c.html#`

1. Lancer `frama-c-gui first-example.c`, et explorez l'interface. Le centre de l'écran est occupé par le programme normalisé par Frama-C, vous pouvez constater qu'il est légèrement différent du code original (visible dans l'onglet de droite). Sur la gauche, vous pouvez naviguer entre les différents fichiers analysés par Frama-C, et même afficher chaque fonction séparément. L'onglet en bas à gauche vous permet d'appeler différents plugins d'analyse. L'onglet du bas vous donne différentes informations sur le code et les analyses réalisées jusqu'ici.

2. Regardez les résultats des différentes métriques pour le programme analysé. Ces métriques ne sont pas l'objet du cours, mais il est bon de noter que l'outil permet de les afficher.

3. Frama-C vous permet également de surligner les parties du code lisant ou écrivant dans une variable particulière. Pour cela, cliquez sur le nom d'une variable, puis sur "Occurences". Toutes les occurences de la variables ainsi pointés seront alors mises en surbrillance. Dans l'onglet "Occurences" de gauche, vous pourrez séléctionner uniquement les occurences qui lisent (Read) ou écrivent (Write) dans la variable en question. Cette analyse fait appel à Eva, un module que nous n'utiliserons pas (cf, plus bas).

4. Réinitialiser le fichier en cliquant sur le raccourci correspondant dans les icônes dans la barre des menus – alternativement relancez frama-c. Remarquez que les trois fonctions du programme disposent d'une spécification. Demandez à Frama-C de prouver ces spécifications en cliquant avec le bouton droit sur le nom des fonctions puis sur "Prove function annotation by WP". Notez qu'une coche verte (ou un autre symbole en fonction de votre thème de couleur) apparaît à coté du contrat dans les 3 cas, montrant que la spécification a été prouvée. Dans l'onglet "WP goal" de l'écran du bas, vous pouvez avoir un résumé de ce qui a été prouvé et par quel prouveur. Dans ce cas, c'est Qed qui a réussi la preuve, et vous pouvez noter que la propriété qu'il avait à démontrer était simple. Il est possible d'obtenir le même résultat en lançant Frama-C avec l'option `-wp`.

5. Vous semble-t'il normal que la spécification ait été prouvée ? Pouvez-vous trouver des exemples d'exécutions où une de ces fonctions échouera ?

6. En réalité, WP suppose qu'aucune erreur d'exécution ne se produit lorsqu'il cherche à prouver un contrat de fonction. Il est cependant capable de générer des assertions correspondant à l'absence de telles erreurs. Réinitialisez à nouveau Frama-C. Cliquez avec le bouton droit sur le nom d'une fonction, puis sur «insert wp-rte guards», puis redemandez à Frama-C de prouver les contrats des fonctions. Vous noterez que les assertions générées ne sont pas prouvées, et que les contrats sont maintenant prouvés sous condition que ces nouvelles assertions le soient. Ces assertions peuvent être générées au lancement de Frama-C avec l'option `-rte` (pouvant être couplée avec les précédentes).

7. La version de Frama-C installée au Cremi est la version Cobalt (27). La documentation se trouve à ce lien[4]. Vous pourrez vous y reporter en cas de doute.

À noter qu'il existe un outil de frama-c dont nous n'avons pas discuté ici, Eva, qui réalise une analyse appelée interprétation abstraite sur le code. Cette fonctionnalité n'est

[4] `https://www.frama-c.com/fc-versions/cobalt.html`

utile que sur des programmes complets (du moins, dans son implémentation actuelle) et ne sera pas utilisée dans le cadre de ce cours. Cet outil est plus utile pour explorer un programme inconnu, et donne plutôt des garanties sur des exécutions particulières. Il sera néanmoins lancé si vous regardez les occurrences d'une variable (pensez à réinitialiser le fichier avant de lancer WP si vous le faites).

Exercice 2: Premier contrat de fonction

Point technique:

Un contrat de fonction se note en commentaire dans l'en-tête de la fonction. Pour être reconnu par Frama-C, le commentaire doit commencer par un `@` (juste après le `/*` ou le `//`).

Un contrat peut contenir une ou plusieurs clauses `ensures`, indiquant une propriété devant être vraie *à la sortie* de la fonction. Cette clause (entre autres) peut mettre en relation les arguments de la fonction (sans précision, le contrat parlera de leur valeur lors de l'appel de la fonction), et la *valeur de retour* de la fonction avec le mot clé `\result`. Il est également possible d'y utiliser des variables et des constantes globales. Leur valeur sera évaluée dans le contexte de retour de la fonction.

Exemple: `/*@ ensures \result == a + b;*/`

Nous verrons rapidement plus de constructions.

Point technique:

Un contrat de fonction peut être placé dans un fichier header (.h). C'est ce que nous ferons dans la plupart des cas (notamment dans cet exercice), notamment pour ne pas modifier des fichiers de code que l'on souhaite spécifier. Il est même possible de placer les spécifications dans un fichier non directement inclus dans le code à vérifier (en l'indiquant à Frama-C) pour importer la spécification dans un code préexistant sans modifier ses fichiers .h directement.

1. Donnez une spécification de la fonction codée dans le fichier max.c, que vous placerez dans le fichier max.h. Vérifiez avec Frama-C que la fonction satisfait votre spécification.

2. Vérifiez que votre spécification est complète en vérifiant que les codes max_wrong1.c et max_wrong2.c ne satisfont pas votre spécification.

Point technique:

Il y a plusieurs manières d'écrire une spécification pour une fonction. Néanmoins, certaines sont à mon avis moins commodes et moins informatives – et également moins extensibles à des cas généraux. Une règle générale est selon moi que les implications devraient être réservées à des comportements très différents des fonctions et non à des cas qui en réalité décrivent une même fonction (typiquement, un maximum peut s'exprimer par des propriétés sur le résultat). La raison profonde, c'est que les implications (ou, en expression informelle, les phrases du style si ... alors ...) c'est plus dur à comprendre pour un humain qu'une propriété globale. Évidemment, il y a des cas où on ne

. coupera pas aux implications.

3. Spécifiez maintenant la fonction dans max_5.c (évidemment, c'est une généralisation de la spécification précédente). Votre spécification devrait être similaire à la précédente et pas beaucoup plus longue (si vous n'avez pas d'implications – si vous avez des implications, vous devriez comprendre pourquoi c'est une mauvaise idée).

Exercice 3: Préconditions

Point technique:

Une *précondition* est une propriété que l'on suppose vraie à l'entrée de la fonction. Les preuves seront faites en supposant que la propriété est vraie. Bien sûr, pour que le programme soit correct, toute fonction appelante doit respecter cette précondition (et donc, il faut la prouver là).

La précondition se note en ACSL avec le mot-clé `requires`. Les clauses `requires` doivent être placées *avant* les clauses `ensures` dans un contrat de fonction. Elles ne peuvent pas parler de `\result` (puisqu'il n'existe pas avant l'appel) et les variables qui y sont utilisées seront évaluées dans le contexte d'appel de la fonction.

Exemple: `/*@ requires a + b >= 3;*/`

1. Observez que la spécification de plus_one n'est pas satisfaite.
2. Écrivez une précondition qui la rend vraie. Pour la trouver, vous pouvez appliquer le calcul de weakest precondition défini en cours (peut-être pas encore pour le moment, mais c'est dans le poly). Attention ici, on ne vous demande de laisser la post-condition telle qu'elle vous est fournie (ce n'est pas une bonne post-condition, mais c'est un détail).
3. Rappelez Frama-C avec l'option `-rte`, et observez que l'assertion générée n'est pas satisfaite.
4. Écrivez une seconde précondition qui rend l'assertion générée par RTE vraie.
5. Faites de même avec la fonction div.
6. Vérifiez, grâce au fichier `calling-functions.c` (en prouvant les assertions de ce fichiers) que vos préconditions sont satisfaites sur les good_call, et non-satisfaites sur les bad_call. Vous pouvez, grâce à une fonction main que vous implémenteriez observer le résultat produit par gcc sur les bad_call.

Point technique:

Les bad_call (à part le premier) de l'exemple précédent sont des exemples d'appel où le comportement de l'addition et de la division ne sont pas spécifiés par la norme du C. Cela signifie que les compilateurs compilant du C peuvent adopter n'importe quel comportement pour ces appels en respectant la sémantique du C. De tels appels ne devraient donc pas apparaître dans un programme : ici, vous pouvez voir ce que gcc fait sur ces appels,mais d'autres compilateurs peuvent faire d'autres choix, et des versions ultérieures de gcc

pourraient changer ces choix. De même des optimisations aggressives peuvent changer le comportement de ce code tout en respectant la norme du C.

Exercice 4: Un autre contrat de fonction

1. Écrivez un contrat de fonction pour la fonction présente dans affine.c. Ce contrat devra permettre de prouver en incluant les assertions générées par RTE.

Exercice 5: Prédicats et comportements

1. Écrivez un contrat de fonction pour la fonction caseResult du fichier result-case.c. On rappelle à toutes fins utiles qu'un «si ... alors ...» correspond généralement à une implication en logique, et que le «sinon» n'existe pas. On rappelle également qu'il n'y a pas d'ordre d'évaluation des formules (ça n'a juste pas de sens, ce n'est pas un programme).

   Dans cet exercice, le résultat correspond à un certain ordre entre les différents arguments. Pour chaque résultat possible, identifiez la relation entre les différents arguments et écrivez-la la plus simplement possible. Vous devriez avoir trois cas très semblables.

2. Votre spécification contient certainement un certain nombre de formules très similaires, et est de ce fait peu lisible. Nous allons régler cela à l'aide de prédicats.

Point technique:

Un *prédicat* dans Frama-C peut simplement être vu comme une macro logique. Son principal intérêt est de rendre les spécifications lisibles (notamment en n'écrivant qu'une seule fois des sous-formules complexes et en leur donnant un nom explicite).

Exemple : `/*@ predicate inInterval(integer a, integer b, integer c) = a >= b && a <= c;*/` est un prédicat disant que a est inclus dans l'intervalle [b,c]. `inInterval(1,0,3)` est vrai, alors que `inInterval(5,1,4)` est faux.

Note: les prédicat peuvent avoir des arguments de n'importe quel type existant. Nous verrons plus d'exemples par la suite.

   En vous inspirant de l'exemple de syntaxe fourni dans predicate-example.c, définissez deux prédicats permettant de simplifier les parties gauches de vos implications dans votre spécification.

3. Vous pouvez également remarquer que votre spécification dispose de 4 cas disjoints dans son comportement, mais que, écrite comme elle l'est, ce n'est pas si visible.

Point technique:

Les *comportements* permettent de spécifier des contrats de fonctions valides uniquement sur certaines plages de données. Cela permet essentiellement de rendre lisible des contrats de fonctions dont le comportement est très différent entre plusieurs cas.

Concrètement, un comportement est d'abord nommé avec la clause `behavior`

> `toto:`, puis on peut lui mettre une ou plusieurs clause `assumes F;` (ou F est une formule logique) qui désignent le domaine du comportement, et ensuite de clauses `requires` et `ensures` qui ont le même rôle que dans un contrat classique.
>
> On peut enfin ajouter, dans le cas d'un contrat avec comportements, deux clauses permettant de vérifier que le contrat est bien formé : `disjoint behaviors` qui est vraie si les domaines comportements sont disjoints, et `complete behaviors` qui est vraie si les comportements couvrent tous les cas possibles des données. Ces deux clauses peuvent être appelées sur certains comportements seulement (ex. `disjoint behaviors A,B,C;` sera vraie si les comportement A, B et C sont disjoints).

En vous inspirant de l'exemple de syntaxe fourni dans behavior-example, donnez une spécification avec 4 comportements (et utilisant vos prédicats). Vos ensures devraient être de la forme `\result == a`. Vous vérifierez également que vos comportement sont disjoints et couvrent tous les cas possibles.

Les deux techniques présentées dans cet exercice sont simplement des syntaxes alternatives de ce que l'on saurait déjà écrire, de manière à le rendre plus lisible. Cela ne devrait pas changer le comportement des solveurs, et est une bonne pratique à encourager. Mais en pratique, il y a des fois des blagues dans les preuves avec cela, et tout n'est pas aussi simple que ça à exprimer sous cette forme. Il ne faut donc pas absolument bannir les implications, il vaut mieux réserver les comportements pour de véritables comportements disjoint de la fonction. À noter également qu'on ne peut pas mettre des comportements dans des comportements.

Exercice 6: Produire du code sûr

1. Implémentez les fonctions présentes dans `exo6.h` en respectant leurs spécifications informelles. Vous prendrez soin de donner des contrats de fonction, avec des clauses `requires` si nécessaire, qui seront prouvés par WP lorsque l'on inclue les gardes générées par RTE. Vous pourrez utiliser des prédicats et des comportements si besoin (pour la deuxième, cela vous permettra des clauses requires plus fines). Bien évidemment, n'utilisez que des variables de type int pour votre code (si vous mettez des flottants, vous n'arriverez pas à prouver quoi que ce soit).