Spécification et preuve formelle de programmes : Frama-C et boucles (TD2)

Frama-C

Bon, faire des preuves à la main, c'est rigolo deux minutes, mais c'est clairement très long, répétitif et source d'erreurs. Heureusement, comme le calcul de la weakest precondition est mécanique, on peut facilement l'automatiser et c'est exactement ce que Frama-C fait pour nous [a]. Il se trouve que Frama-C n'est pas disponible directement sur les machines de l'ENSEIRB, mais il y a deux solutions :

- Installer Frama-C sur votre ordinateur personnel (Linux, MacOS ou Windows via WSL)
- Utiliser le fichier `frama-c.sh` fourni dans `TD2/src`.

La solution n°2 est un script que vous pouvez exécuter sur les machines de l'ENSEIRB, et cela lancera Frama-C. Vous pouvez librement passer des arguments à ce script comme si vous appeliez Frama-C directement. Pensez à rendre le script exécutable avec la commande `chmod +x frama-c.sh`.

Pour ce qui est de la solution n°1, vous pouvez suivre les instructions suivantes (pour Linux) [b] [c] :

- Installer OPAM (le gestionnaire de paquets d'OCaml) avec la commande `sudo apt install opam`;
- Initialiser OPAM avec la commande `opam init -compiler 4.14.1`
- Mettre à jour l'environnement shell avec la commande `eval $(opam env)` [d];
- Installer Frama-C avec la commande `opam install alt-ergo z3 why3 frama-c`;
- Mettre à jour l'environnement shell avec la commande `eval $(opam env)`;
- Laisser Why3 se configurer avec la commande `why3 config detect`.

Si tout se passe bien, après ces commandes, vous aurez Frama-C installé sur votre machine, vous pouvez tester en utilisant la commande `frama-c-gui`.

Une question légitime est : "*qu'est-ce que c'est que alt-ergo, z3 et why3 ?*" Ce sont des *provers*, c'est-à-dire des outils qui permettent de prouver automatiquement des formules logiques. Frama-C utilise ces outils pour prouver les obligations de preuves générées par le calcul de weakest precondition. Ce n'est pas très important de savoir comment ils fonctionnent, concrètement, ils s'occupent du calcul de la weakest precondition à notre place, et s'assurent que la précondition implique bien la weakest precondition calculée. Dans Frama-C, vous pouvez choisir d'utiliser Alt-Ergo ou Z3 comme prover (en cliquant sur `Provers...` au milieu à gauche). Dans la plupart des cas, le plus optimal est d'utiliser les deux.

Voyons comment utiliser Frama-C à présent. Dans le dossier `TD2/src`, vous trouverez un fichier `example.c`. Dans un premier temps, ouvrez-le avec un éditeur de texte. Vous y trouverez différentes fonctions, annotées avec des commentaires décrivant des spécifications en ACSL (le langage de spécification utilisé par Frama-C). Lorsque Frama-C analyse un fichier, il considère que tout commentaire commençant par `/*@` [e] est une annotation ACSL (et donc une spécification). Dans le cas de la fonction `add_one`, on remarque qu'il y a écrit : `requires x < INT_MAX;`, c'est notre précondition. Le mot clé `requires` indique que la formule donnée ci-après, jusqu'au point-virgule, est une précondition. De même, on remarque le mot clé `ensures`, qui indique que la formule qui suit est une postcondition. Ici, on spécifie que le résultat de la fonction doit être égal à `x + 1`. Il y a aussi le mot clé `assigns`, qui indique quelles variables peuvent être modifiées par la fonction (cela permet de décrire les effets de bord si des pointeurs sont utilisés). Dans le cas de cette fonction, il n'y a pas d'effet de bord, donc on écrit `assigns \nothing;`.

À présent, ouvrez le même fichier avec Frama-C, via la commande `frama-c-gui example.c`, vous allez voir l'interface principale de Frama-C. Sur la gauche, vous avez la liste des fonctions du code, vous pouvez les observer séparément en cliquant dessus, ou toutes en même temps en cliquant sur le nom du fichier tout en haut. Vous remarquez peut-être que le code n'est pas exactement le même que celui écrit dans le fichier, c'est normal. Frama-C effectue une transformation du code source en un code intermédiaire appelé *C Intermediate Language* (CIL), qui est plus facile à analyser. On va demander à Frama-C d'analyser la fonction `add_one`, afin de prouver qu'elle respecte bien sa spécification. Pour cela, vous pouvez faire un clic droit sur le nom de la fonction et sélectionner *Prove function annotations by WP*. Cela va demander à Frama-C de calculer la weakest precondition de la fonction `add_one` et de prouver que la précondition

implique cette weakest precondition. Après quelques instants, vous devriez voir apparaître des coches vertes, ces coches vertes indiquent que la spécification a été prouvée avec succès.

Faisons de-même avec la fonction `div`. Encore une fois, vous devriez voir des coches vertes apparaître. Super ..., sauf que cette fois, la fonction contient en fait un bug. En effet, si `y = 0`, alors on a une division par zéro, et le comportement de la fonction est indéfini (en pratique, le programme va planter). Pourquoi Frama-C n'a-t-il pas détecté ce problème ? Cela vient du fait que Frama-C, par défaut, ne vérifie pas les erreurs d'exécution (les overflows, les divisions par zéro, ...). Donc si le code contient des erreurs d'exécution, Frama-C ne les détectera pas. Il faut donc lui demander explicitement de vérifier ces erreurs. En théorie, il faudrait ajouter des annotations à chaque ligne de code qui pourrait produire une telle erreur. Heureusement, détecter ces lignes problématiques est très facile, et Frama-C permet de le faire automatiquement. Pour ce faire, on va d'abord réinitialiser l'analyse en cliquant sur la flèche circulaire en haut à gauche (on peut aussi relancer Frama-C). À présent, en faisant un clic droit sur la fonction `div`, on peut sélectionner `Insert wp-rte guards`. En faisant cela, Frama-C va analyser la fonction `div` et insérer automatiquement des annotations ACSL pour vérifier les erreurs d'exécution. Si on essaie à nouveau de prouver la fonction `div` avec `Prove function annotations by WP`, on remarque que cette fois, certaines annotations n'ont pas pu être prouvées (une coche orange apparaît). De plus, on remarque que la postcondition reçoit une coche verte-orange. Cela signifie qu'elle est vraie si certaines autres annotations, qui n'ont pas pu être prouvées (donc qui ont une coche orange), sont vraies. Dans ce cas, il faut comprendre que la spécification sera correcte si les gardes rajoutées sont vraies aussi. Pour l'instant ce n'est pas le cas, mais en rajoutant la bonne précondition, on peut faire en sorte que tout soit prouvé. Ici, on voit deux annotations qui n'ont pas pu être prouvées :

- La première indique que `y` ne doit pas être égal à zéro avant la division.
- La deuxième indique que le résultat de la division ne doit pas causer un overflow.

La première condition est le premier bug qu'on avait remarqué, il faut donc que $y \neq 0$. La deuxième condition est plus subtile, considérons les valeurs $x = INT\_MIN$ et $y = -1$, que se passe-t-il ? Eh bien, en C, le résultat de cette division donne INT_MIN, ce qui n'est clairement pas le résultat attendu. On doit donc modifier la précondition pour s'assurer que les deux valeurs ne peuvent pas être utilisées en même temps.

---

a. Cet outil fait bien plus que ça, mais nous, on va s'en servir pour faire les calculs de weakest precondition à notre place.

b. Les commandes données ici sont aussi listées dans le fichier `TD2/src/INSTALL.md`, ce qui permet de plus facilement copier-coller les commandes

c. Si vous n'êtes pas sous Linux, je vous invite à aller sur la page officielle de Frama-C, je n'ai aucun idée de comment ça se passe sous MacOS ou Windows.

d. Je vous conseille **très fortement** d'ajouter cette commande à votre fichier `.bashrc`, sinon il faut la relancer à chaque fois que vous ouvrez un nouveau terminal.

e. *Attention : il ne faut pas d'espace avant l'arobase.*

Exercice 1

Modifier le fichier `example.c` pour ajouter la bonne précondition à la fonction `div`, et tester à nouveau la preuve avec Frama-C (pensez à bien insérer les gardes avec `Insert wp-rte guards` avant de prouver). Faites le même test pour la fonction `add_one`, et modifier la précondition si nécessaire.

Pour gagner du temps, vous pouvez lancer Frama-C en lui demandant en avance d'ajouter les gardes et de faire les preuves de toutes les annotations. Pour ce faire, utilisez la commande `frama-c-gui -wp -wp-rte -wp-prover z3,alt-ergo <fichier>`.

Exercice 2

Écrire la spécification ACSL des fonctions `mod` et `mod2` du fichier `TD1/src/ex1.c` et vérifier la correction de ces fonctions avec Frama-C.

Exercice 3

Écrire la spécification ACSL de la fonction `max` du fichier `TD1/src/ex2.c` et vérifier la correction de cette fonction avec Frama-C.

Exercice 4

Écrire la spécification ACSL de la fonction `div` du fichier `TD1/src/ex4.c` et vérifier la correction de cette fonction avec Frama-C.

Exercice 5

Écrire la spécification ACSL de la fonction `swap` du fichier `TD1/src/ex3.c` et vérifier la correction de cette fonction avec Frama-C.

Vous aurez sûrement envie de préciser que les pointeurs passés en argument sont *valides* (typiquement non nuls [a]). Pour décrire cette propriété, vous pouvez utiliser l'annotation `\valid(...)` qui permet de garantir qu'un pointeur est valide.

a. En réalité, un pointeur peut être invalide sans être nul, par exemple après un free.

Exercice 6

Ouvrez le fichier `TD2/src/ex5.c`, dedans vous trouverez la fonction `sum`. Cette fonction calcule la somme des entiers de 0 à n. Dans le fichier, la spécification est déjà donnée.

1. Tenter de prouver la spécification avec Frama-C. Que constatez-vous ?

En fait, la méthode que nous avons vue pour calculer la weakest precondition ne fonctionne pas telle quelle pour les boucles (et c'est bien normal, nous n'avons pas vu comment traiter le cas des boucles dans nos règles de calcul de weakest precondition). Dans les faits calculer la weakest precondition d'une boucle est pénible. Ce qui pose problème, c'est que pour calculer l'effet d'une boucle, il faudrait savoir combien de fois la boucle va s'exécuter, or ce n'est pas forcément évident à déterminer statiquement (c'est même en général impossible). Pour résoudre ce problème, on aimerait bien avoir une sorte de "résumé" de la boucle, qui nous dit, sans avoir à dérouler toute la boucle, quel est son effet. Par exemple, dans la fonction `sum`, on aimerait bien dire : "*après la boucle, la variable* `res` *vaut la somme des entiers de* 0 *à* n".

Pour ce faire, on va utiliser des **invariants de boucle**. Un invariant de boucle est une propriété qui doit être vraie :

- Au début de chaque itération de la boucle ;
- A la fin de chaque itération de la boucle ;
- Après la fin de la boucle.

Ainsi, lors du calcul de la weakest precondition, on pourra utiliser cet invariant pour résumer l'effet de la boucle, puisque cet invariant est en particulier vrai au début de la boucle (ce qui permet de calculer la weakest precondition avant la boucle). La partie compliquée est de trouver le bon invariant de boucle, qui permet de prouver la spécification. En effet, un invariant de boucle possible pour la fonction `sum` est $0 \leq i \leq n+1$. C'est un invariant tout à fait correct, mais il n'est pas assez fort pour prouver la spécification. On a besoin d'un invariant qui permet d'impliquer la postcondition à la fin de la boucle.

2. Afin d'ajouter un invariant de boucle dans Frama-C, on utilise l'annotation `loop invariant <formule>;` juste avant la boucle. Un exemple est donné dans le fichier `ex5_weak_invariant.c`. Notons qu'en plus de l'invariant, il est nécessaire d'indiquer les variables modifiées par la boucle avec l'annotation `loop assigns <liste_de_variables>;`.

Essayer de trouver des invariants suffisamment forts pour prouver la spécification de la fonction `sum` avec Frama-C. Rappelez vous qu'un invariant doit être vrai au début et à la fin de chaque itération de la boucle, ainsi qu'après la fin de la boucle.

Exercice 7

1. Écrire la spécification ACSL de la fonction `is_sorted` du fichier `TD1/src/ex6.c` [a].

Note : Vous aurez sûrement besoin d'un quantificateur, en ACSL, on peut écrire $\backslash forall\ integer\ i; 0 <= i < n ==> i == size$; *pour exprimer "pour tout* i *entre* 0 *et* $n-1$, i *est égal à* $size$*".*

2. Proposer des invariants de boucle permettant de prouver la correction de cette fonction avec Frama-C.

a. Un tableau est une séquence de pointeurs, donc l'utilisation du mot clé `\valid` est nécessaire pour garantir la validité des pointeurs.

Exercice 8

1. Expliquer ce que fait la fonction `mystery` dans le fichier `TD2/src/ex7.c`.

2. Écrire la spécification ACSL de cette fonction.

3. Proposer des invariants de boucle permettant de prouver la correction de cette fonction avec Frama-C.