CentraleSupélec

TP Frama-C / WP – le 16 janvier 2018

Nikolaï Kosmatov

L'objectif de ce TP est l'utilisation du langage de spécification ACSL et du greffon WP de Frama-C pour la spécification et la preuve des programmes C. Un bref compte-rendu (au format PDF ou TXT) devra être envoyé par mail à Nikolai.Kosmatov@cea.fr à la fin de la séance. Merci de recopier les fichiers C complétés directement dans ce compte-rendu si nécessaire (ne pas soumettre d'archive ou de fichiers C séparés).

Notions de ACSL (ANSI C Specification Languge):

- Les spécifications (annotations) ACSL sont des commentaires C spéciaux de deux types :
  `/*@ … */`
  `//@ …`
- Annotations de bases pour spécifier une fonction :
  `requires <condition>;` donne une précondition (supposée vraie)
  `ensures <condition>;` donne une postcondition (à prouver)
  `\result` fait référence à la valeur retournée par la fonction
  `\old(var)` fait référence à la valeur de var en entrée de la fonction.
- La postcondition et la précondition se terminent par un ";" et sont optionnelles, mais si présentes en même temps, elles doivent apparaître dans le même commentaire et la précondition doit précéder la postcondition
- Les conditions (formules logiques) de base sont définies comme suit :

```
formule ::= expr
| expr rel expr
| formule ==> formule
| formule <==> formule
| formule && formule
| formule || formule
| \forall type ident ; formule
| \exists type ident ; formule
rel ::=  == | != | < | <= | > | >=
```

- La validité des locations mémoire peut être exprimée par la condition suivante :
  pour un pointeur T *p; vers une seule case du type T :
  `\valid( p )`
  pour un pointeur T *p; vers un tableau de taille n de données de type T :
  `\valid( p + (0..n-1) )`
- On trouvera la description exhaustive du langage ACSL dans le manuel (présent aussi dans le répertoire frama-c/doc/manuals/) nommé
  `acsl-implementation<version>.pdf`

Exercice 1.

Nous allons spécifier et prouver le programme du fichier ex1.c. Pour lancer le greffon de preuve WP, on écrira la commande

```
frama-c-gui -wp ex1.c
```

Si le programme est syntaxiquement correct, cela lance l'interface de Frama-C et WP essaie de produire les preuves. Le panneau latéral liste les fichiers ouverts, un clic sur le nom du fichier permet de voir son contenu, la flèche juste à côté du nom permet de dérouler la liste des fonctions

présentes dans le fichier.
Le statut de preuve pour chaque obligation de preuve sera indiqué (par ex., prouvé ou échec).
Le panneau inférieur contient divers onglets, celui intitulé « WP Goals » permet de voir plus précisément le statut de chaque objectif de preuve (but) individuellement. Un clic sur un but indique, par surlignage dans le code, à quelle propriété il fait référence.

1. Lancez la preuve pour le fichier ex1.c . Est-il prouvé ?
2. Pourquoi la spécification de la fonction maxint n'est pas correcte ? Commentez le corps de la fonction et donnez une autre version de la fonction qui satisfait cette spécification, mais ne retourne pas le maximum de ses deux entrées x et y. Vérifiez que cette version erronée passe aussi la preuve.
3. Corrigez la spécification et refaites la preuve du code initial (correct). La version erronée de la question précédente passe-t-elle la preuve maintenant ? Donnez le programme corrigé complet et expliquez la correction.
4. Conclusion : expliquez l'importance de la précision de la spécification.

Exercice 2.

Nous allons spécifier et prouver le programme du fichier ex2.c.

1. Lancez la preuve sur le fichier. Est-il prouvé ?
   Des vérifications supplémentaires :

- Si on regarde dans le panneau inférieur, l'onglet « Messages » indique que WP nous a transmis un message qui est :
  « Missing RTE Guards »
  Ce message nous dit qu'aucune vérification d'erreur d'exécution n'a été effectuée par WP, et que l'on raisonne uniquement dans le monde de la logique, sans prendre en compte les spécificités de C.
- Pour produire les vérifications en question, il est possible de demander à WP de les générer au lancement de Frama-C par l'intermédiaire de la ligne de commande (que vous utiliserez systématiquement par la suite) :

```
frama-c-gui -wp -wp-rte fichier.c
```

2. Relancez la preuve pour le fichier. Pourquoi n'est-il pas complètement prouvé ? Le risque de dépassement est-il réel ?
3. Ajoutez une précondition limitant *n* aux valeurs entre -100 et 100. Refaites la preuve et vérifiez qu'elle passe. Donnez le programme corrigé complet.

Exercice 3.

Nous allons spécifier et prouver le programme du fichier ex3.c.

1. Lancez la preuve pour le fichier (sans oublier les RTEs). Pourquoi n'est-il pas prouvé ?
2. Ajoutez une précondition assurant la validité du pointeur *p*. Refaites la preuve et vérifiez qu'elle passe. Donnez le programme corrigé complet.

Notions de ACSL : assigns

- Pour préciser les variables (locations mémoire) que la fonction a le droit de modifier en dehors de ses variables locales, on utilise la clause :

```
assigns v1, v2, ..., vN ;
```

ou, si la fonction ne doit modifier aucune variable non locale :

```
assigns \nothing ;
```

Exercice 4.

Dans cet exercice, on souhaite montrer qu'une version erronée du programme ne doit pas être prouvée avec une spécification correcte et complète. Nous allons compléter la spécification et prouver le programme du fichier ex4.c. Les fichiers ex4_error*.c contiennent des versions erronées du même programme. Notre objectif sera donc de trouver une **unique spécification** qu'on recopiera dans toutes les versions afin de discriminer les versions erronées ex4_error*.c de la version correcte ex4.c.

1. Examinez les trois versions erronées du fichier ex4.c qui modifient des variables en plus ou ne font pas les mêmes actions. La preuve passe-t-elle pour ces versions erronées ?
2. Précisez la postcondition et spécifiez les variables pouvant être modifiées par la fonction. On rappelle que **la même spécification** sera utilisée pour toutes les versions. Refaites les preuves. Donnez le programme correct complet. Expliquez vos modifications.
3. Conclusion : pourquoi faut-il spécifier les variables modifiées ?

Notions de ACSL : séparation.

- Il est possible de préciser que « n » pointeurs concernent des zones mémoires différentes par l'intermédiaire du prédicat :

```
\separated(p1, ..., pN)
```

De cette manière on peut exprimer que les différents pointeurs listés concernent des zones mémoires différentes.

- Comme pour \valid, il est possible de spécifier des intervalles d'adresses :

```
\separated(p1+(0..n1), ..., pN+(0..nN))
```

Exercice 5.

Nous allons spécifier et prouver le programme du fichier ex5.c.

1. Lancez la preuve du fichier ex5.c, est-il prouvé ? Pourquoi ?
2. Corrigez la spécification, expliquez vos modifications et donnez le fichier corrigé.

Notions de ACSL : invariant, variant et assigns de boucles.

- L'invariant de boucle permet la preuve à travers les boucles sans forcément connaître le nombre d'itérations. En général, il exprime la partie de la propriété à prouver après la boucle qui devient vérifiée après les *n* premières itérations. Il doit être donné juste avant la boucle. Il s'écrit :

```
loop invariant <condition> ;
```

- Le variant de boucle permet de prouver la terminaison de la boucle sans forcément connaître le nombre d'itérations (on parle alors de la correction totale). Il doit :
  - être une expression, fonction des variables du programme,
  - être positif lorsqu'on commence une itération (c'est-à-dire lorsque la condition de la boucle est vérifiée et lorsque l'invariant de la boucle est vérifié),
  - être strictement décroissant à chaque itération.

    Comme il ne peut diminuer infiniment, la terminaison de la boucle en découle. Le variant s'écrit :

```
loop variant <expression> ;
```

- Pour préciser les variables, y compris locales, que la boucle a le droit de modifier, on utilise la clause :

```
loop assigns v1, v2, ..., vN ;
```

  Si la fonction n'a rien le droit de modifier (cas extrêmement rare), il est possible, comme pour l'assigns de fonction, de préciser « \nothing ». (Cependant cela voudrait dire que l'on ne peut pas poser de variant à cette boucle).
- L'ordre des clauses pour la boucle **doit** être : loop invariant, loop assigns, loop variant.
- Un exemple complet est donné par le fichier getMin.c avec une fonction qui retourne le minimum du tableau donné t de la taille donnée n. Faites la preuve.

Exercice 6.

Nous allons spécifier et prouver uniquement la terminaison des fonctions du fichier ex6.c.

1. Complétez **uniquement** les variants de boucles. Faites les preuves et vérifiez qu'elles passent. Donnez vos variants de boucles et expliquez votre choix.

Exercice 7.

Nous allons spécifier et prouver le programme du fichier getMinSubarray.c. La fonction getMinSubarray est une version de getMin qui cherche le minimum dans le sous-tableau des indices *k*..*n*-1 du tableau donné *t* de la taille donnée *n* et prend l'indice *k* en entrée en plus.

1. Complétez la spécification du programme. Vous pouvez vous inspirer de getMin.c. Faites la preuve et vérifiez qu'elle passe. Donnez le programme complet.
2. Vous avez effectué la preuve de la correction totale (le programme termine et il vérifie la spécification) qui est plus forte que la correction partielle (si le programme termine, alors il vérifie la spécification). Quelle partie de la spécification permet de prouver la terminaison ?

Exercice 8.

Nous allons spécifier et prouver le programme du fichier all-zeros.c qui retourne 1 si tous les éléments dans le tableau donné t de la taille donnée n sont nuls, et 0 sinon.

1. Complétez la spécification du programme. Faites la preuve et vérifiez qu'elle passe. Donnez le programme complet.
2. Quelle partie clef de la spécification permet de prouver la postcondition pour un nombre quelconque d'itérations ?

Notions de ACSL : énoncés intermédiaires.

- Parfois un prouveur automatique peut ne pas réussir à prouver une propriété. Différentes possibilités d'indiquer une propriété intermédiaire (un axiome, un lemme, une assertion) existent dans le langage ACSL pour l'aider. Elles sont décrites dans le manuel. Si ce résultat intermédiaire lui-même n'est pas prouvé par le prouveur, on devra s'assurer par ailleurs de sa validité (par exemple, en le prouvant dans Coq).

Exercice 9.

Nous allons spécifier et prouver le programme du fichier binary_search.c qui effectue une recherche binaire d'un élément donné (*query*) dans un tableau trié donné (*arr*) de la taille donnée (*length*) et retourne -1 si cet élément n'est pas trouvé, et son indice sinon. Pour cette preuve, nous aurons besoin de donner au prouveur automatique un lemme sur la division par 2 qui peut être prouvé par ailleurs si le prouveur n'arrive pas à le prouver, et une assertion supplémentaire (que le prouveur arrive à prouver) au milieu de la fonction pour diriger la preuve.

1. Complétez la spécification du programme. Faites la preuve et vérifiez qu'elle passe. Donnez le programme complet.
2. Un bug connu est dû à une version erronée de calcul de mean, ici en commentaire. Faites la preuve en utilisant cette version. Que constatez-vous ? Ce bug aurait-il pu rester inaperçu si un outil de preuve était systématiquement appliqué ?

Exercice 10.

Nous allons spécifier et prouver le programme du fichier max_abs.c. Selon la spécification, la fonction max_abs retourne le maximum des valeurs absolues de ses deux arguments.

1. Considérer la première version max_abs1.c. Pouvez-vous expliquer l'échec de la preuve ? Donner le programme corrigé qui passe la preuve et expliquer la correction.
2. Considérer la deuxième version max_abs2.c. Pouvez-vous expliquer l'échec de la preuve ? Donner le programme corrigé qui passe la preuve et expliquer la correction.
3. Pouvez-vous expliquer pourquoi un échec de preuve ne se produit pas toujours dans la fonction dont le contrat doit être corrigé. Justifier ce phénomène par la démarche de la vérification modulaire.
4. (sans réponse écrite) Vous pouvez également expérimenter en retirant différentes parties des contrats et en essayant de prédire où la preuve va échouer avant de lancer WP. Il peut être également utile de faire cet exercice après avoir rajouté la définition des fonctions max et abs.