# Conception Formelle

2018-2019

## TP1: Découverte de Frama-C

Vincent Penelle

---

**Points abordés**

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

---

Récupérer l'archive suivante[1] et décompressez-la.

### Exercice 1: Introduction à Frama-C

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 et dans l'onglet "Studia", essayez "Reads" et "Writes".

   Erratum: j'ai écrit ce sujet en regardant une version plus récente de Frama-C. Vous n'avez pas ça sous Silicon. Vous avez cependant l'onglet "Occurence" qui surligne toutes les occurrences d'une variable. Vous pouvez dans le volet à gauche sélectionner (dans occurence) sélectionner "Read" et/ou "Write".

4. Lancez l'analyse par valeurs (Value → Run) et inspectez-les valeurs des variables au cours du programme (grâce à l'onglet "Value" dans l'écran du bas). Cette fonction ne sera pas non plus au cœur du cours, et est bien plus riche qu'une simple exécution du flot du programme. Mais vous pourrez l'utiliser pour comprendre un code inconnu, par exemple. Cette analyse peut être exécutée au lancement de Frama-C avec l'option `-val`.

[1] `https://www.labri.fr/perso/vpenelle/Enseignement/ConceptionFormelle18/TP1.tar.gz`

5. Réinitialiser le fichier en cliquant sur la flèche verte en haut à gauche. Remarquez que les trois fonctions du programme dispose 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 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`.

6. Vous semble-t'il normal que la spécification ait été prouvé ? Pouvez-vous trouver des exemples d'exécutions où une de ces fonctions échouera ? Décommentez la ligne annotée `question 6 -- 1`, et relancez Frama-C en effectuant une analyse par valeur. Que constatez-vous ? Faites de même avec `6 -- 2` et `6 -- 3`. Note : des lignes surlignées en rouge dénotent des lignes non-atteintes par le programme.

7. 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. Recommentez la ligne de la question précédente, puis relancez Frama-C. Cochez la case "RTE" de l'onglet "WP", 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).

8. Lancez maintenant une analyse par valeur et observez que les assertions générées par RTE apparaissent maintenant comme étant prouvées. C'est parce que grâce à l'analyse du flot du programme, Frama-C a pu constater que les assertions ne sont jamais violées par ce programme particulier. Cela peut permettre de montrer qu'un programme va bien se comporter même si les fonctions qu'il contient ne sont pas sûres. Toutefois, il est à noter que dans de nombreux programmes, l'analyse par valeurs ne donne pas d'aussi bons résultats et que si jamais le contexte d'appel de la fonction est appelé à changer, il n'y a aucune garantie que tout se passe bien.

9. La version de Frama-C installée au Cremi est la version Silicon (14). La documentation se trouve à ce lien[2]. Vous pourrez vous y reporter en cas de doute.

## 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 `requires`, 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

[2] `http://frama-c.com/download_silicon.html`

globales.

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

Nous verrons rapidement plus de construction.

**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.

### 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).

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

1. Observez que la spécification de plus_one.c n'est pas satisfaite.
2. Écrivez une précondition qui la rend vraie.
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.

### 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: Pointeurs et tableaux

1. Écrivez un contrat de fonction pour la fonction `max_ptr.c` de exo5.c (cf Exercice 2).
2. Affichez les gardes RTE. Que pensez-vous qu'elles indiquent ? Rajoutez-les comme préconditions, et observez que Frama-C considère que tout est correctement prouvé.
3. Même question pour la fonction `sum_first_last`. Attention aux valeurs possibles de $n$.

**Point technique:**

Dans un contrat de fonction, si on veut faire référence explicitement à la valeur d'une variable lors de l'appel de la fonction, on doit la mettre dans la fonction `\old(...)`.

Exemple: `/*@ ensures \old(a[0]) == a[0];*/`

4. Mêmes questions pour les fonction `swap_first_last` et `swap`. Attention, on veut maintenant écrire dans le tableau/pointeur, il doit donc être `\valid` et pas seulement `\valid_read`.

## Exercice 6: Pointeurs ++

1. Écrivez un contrat de fonction pour la fonction `sort_ptr` de `exo6.c`, qui doit permettre de satisfaire les assertions générées par RTE. Attention, il ne doit PAS être satisfait par les implémentations présentes dans `wrong_sort_ptr1.c` et `wrong_sort_ptr2.c`.

2. Écrivez un contrat de fonction pour la fonction `sum_in_pointer`, qui doit permettre de satisfaire les assertions générées par RTE. De plus, vous assurerez que la fonction ne modifie pas la valeur de *b. À votre avis, pourquoi Frama-C n'arrive pas à prouver cette assertion ? Dans quel cas peut-elle être fausse ? (ne trichez pas, essayez de répondre à cette question avant de regarder la suite)

**Point technique:**

Le mot clé `\separated` permet d'assurer que deux pointeurs ne pointent pas vers la même zone mémoire. Il est également possible d'assurer que plusieurs adresses sont disjointes deux à deux grâce au même mot clé.

Exemple: `/*@ \separated(a,b+(0 .. n));*/`.

3. Ajouter la clause `\requires` nécessaire pour que la fonction précédente soit maintenant prouvée.

## Exercice 7: Assigns

1. Exécutez le plugin WP sur le fichier `assigns.c` et observez le résultat produit. Curieusement, les assertions concernant les pointeurs ne sont pas prouvés.

2. Cela est dû au fait que par défaut, Frama-C considère qu'une fonction peut modifier n'importe quelle valeur en mémoire (sauf les variables locales d'autre fonctions. Grâce à l'indication ci-dessous, modifiez les contrats des fonctions `add_one` et `dummy` pour que WP réussisse à prouver toutes les assertions.

3. L'assertion `*b == \old(*b);` n'est toujours pas prouvée. Pourquoi ? (cf exercice précédent)

**Point technique:**

La clause `assigns` permet d'indiquer quelles valeurs la fonction peut être amenée à modifier lors de son exécution. Si elle ne modifie rien, on peut écrire `assigns \nothing`. Attention, WP réussira à prouver tout sur-ensemble des valeurs que la fonction modifie vraiment. Ainsi, une fonction ne modifiant que la valeur `*a` aura les clauses `assigns *a,*b;` et `assigns *a;` prouvées par WP, mais pas la clause `assigns *b;`.

### Exercice 8: Comportements (Behaviors)

1. Lancez WP sur le fichier behavior.c. Frama-c arrive à prouver le contrat de f, mais pas celui de g. Voyez-vous pourquoi ?

**Point technique:**

Un contrat de fonction peut être découpé en différents *comportements* (behaviours) permettant d'affiner le contrat de manière à traiter plusieurs cas séparément. Il contiendra des clauses `assumes` (déterminant à quel cas correspond ce comportement), `requires`, `assigns` et `ensures`, correspondant au contrat de fonction dans ce comportement. Il est toujours possible d'avoir des clauses globales (avant la déclaration des comportements). Notez qu'il est possible de spécifier des `requires` et des `assigns` différents pour chaque comportements.

Exemple:

```
behavior toto:
assumes A;
requires R;
assigns L;
ensures E;
```

Il est également possible de demander que les différents comportements sont *disjoints* et *complets* avec les assertions `disjoint behaviors` et `complete behaviors`.

2. Le contrat n'est pas assez spécifique sur deux points: le résultat dépend de $d$, et cela n'apparaît pas dans le contrat, et les valeurs modifiées dépendent de $d$ aussi. Si cela pourrait être exprimé sans comportement (grâce à des implications), les comportements permettent de l'exprimer de manière plus lisibles. Redéfinissez un contrat de fonction pour f permettant de démontrer le contrat pour g grâce à des comportements.

### Exercice 9: Utiliser Frama-C pour trouver des bugs

1. Lancez Frama-C sur le fichier write.c.
2. Identifiez les bugs grâce à WP et RTE et corriger le code de la fonction writes pour qu'elle satisfasse sa spécification.

### Exercice 10: Produire du code sûr

1. Implémentez les fonctions présentes dans `bonus.h` en respectant leurs spécifications informelles. Vous prendrez soin de donner un contrat de fonction dont les seuls clauses `requires` indiqueront que les pointeurs manipulés sont bien alloués (i.e., `\valid`) et qui seront prouvés par WP lorsque l'on inclue les gardes générées par RTE.