

# Conception Formelle

2021-2022

## 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 24.0 de Frama-c, avec la version 2.4.1 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 4.8.11 est également installée, ce qui vous permettra de l'utiliser également parallèlement à alt-ergo. Toutes ces versions sont les plus récentes en date de janvier 2022.

Pour travailler au CREMI, connectez-y vous grâce à x2go, comme expliqué ici<sup>1</sup>. 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 (vous êtes des informaticiens, quoi).

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<sup>2</sup> 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<sup>3</sup>. N'hésitez pas à consulter les instructions détaillées en cas de problème. Par contre attention, cela installera

---

<sup>1</sup><https://services.emi.u-bordeaux.fr/intranet/spip.php?article125>

<sup>2</sup><https://www.labri.fr/perso/vpenelle/Enseignement/ConceptionFormelle/tp1.tar.gz>

<sup>3</sup><https://www.frama-c.com/html/get-frama-c.html#>

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 mêmes versions qu'au CREMI. Aussi, je vous conseille fortement de, dans la dernière ligne (celle qui installe frama-c) remplacer 'frama-c' par 'alt-ergo.2.4.1 frama-c.24.0'. 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 (une fois x2go lancé) restent nécessaires.

### 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, puis sur "Occurrences". Toutes les occurrences de la variables ainsi pointés seront alors mises en surbrillance. Dans l'onglet "Occurrences" de gauche, vous pourrez sélectionner uniquement les occurrences qui lisent (Read) ou écrivent (Write) dans la variable en question.
4. Lancez l'analyse par valeurs (Eva → Run) et inspectez-les valeurs des variables au cours du programme (grâce à l'onglet "Values" 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`.
5. Réinitialiser le fichier en cliquant sur la flèche verte en haut à gauche. 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 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 prouvée ? 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. 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).
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 Chromium (24). La documentation se trouve à ce lien<sup>4</sup>. 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 *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

<sup>4</sup><https://www.frama-c.com/fc-versions/chromium.html>

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.

### 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.
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 agressives 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).
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 prédicats 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 comportements 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 disjoints 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 `bonus.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).