amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Master informatique
Génie logiciel : TP 5
Codes UE : SINA09EL
Tests : preuve de programme avec Frama-c

Ce travail est à rendre au plus tard le dimanche 22 février à 23:59.

1 Utilisation de Frama-C avec le plugin WP

Nous allons utiliser le logiciel Frama-C, plateforme ouverte et extensible pour l'analyse et la vérification de code C. Cette plateforme dispose de différents greffons (plugins) permettant diverses analyses. Pour la vérification déductive, nous allons utiliser le greffon WP. Ce greffon manipule des formules logiques dont il teste la satisfiabilité. Pour cela, il utilise un ou plusieurs prouveurs. WP vient avec Qed un prouveur interne. Cependant, ils existent des prouveurs externes plus puissants tels que Alt-Ergo, Z3 ou CVC4 également utilisables.

Si vous êtes sur votre machine personnelle, vous pouvez installer Frama-C en suivant les instructions au lien suivant : Instructions d'installation

Si vous êtes sur une machine de l'université, vous pouvez utiliser la version déjà installée de Frama-C. Il vous faut faire quelques étapes de configuration pour pouvoir l'utiliser. Tout d'abord faut configurer la platte-forme why3, qui est un outil de preuve utilisé par Frama-C. Pour cela lancer la commande suivante dans un terminal :

```
why3 config detect
```

Cette commande va détecter les prouveurs installés sur votre machine et les configurer pour qu'ils soient utilisables par Frama-C. Vous devriez voir une liste de prouveurs détectés qui seront rajouté dans la configuration de `why3` (fichier `~/.why3.conf`). Lancer ensuite la commande suivante pour vérifier que les prouveurs sont bien configurés :

```
frama-c -wp-detect
```

Vous devriez obtenir une liste de prouveurs détectés par Frama-C, qui devrait correspondre à la liste obtenue avec la commande précédente.

Une fois que vous avez configurer Frama-C, vous pouvez lancer la commande suivante :

```
frama-c -wp toto.c
```

pour lancer l'analyse de frama-c sur le fichier `toto.c` avec le plugin WP.

Vous pouvez lancer la nouvelle interface graphique de frama-c appelée Ivette avec la commande suivante :

```
ivette -wp toto.c
```

Nous allons au cours des exercices suivants étudier quelques fonctionnalités de base de `frama-c -wp`. Les propriétés que nous donnerons seront écrites en ACSL (ANSI/ISO C Specification Langage).

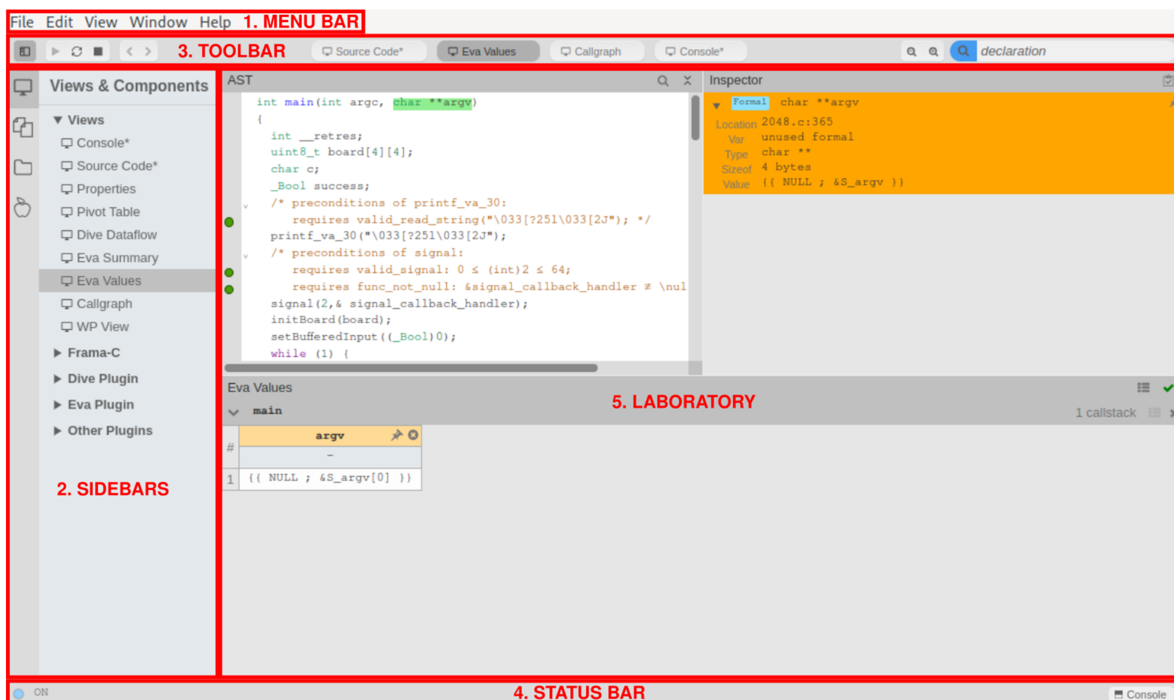
2 Exercice 1

Placez-vous dans le dossier `tp5/Exo1`.

1. Lancer `frama-c` ou `ivette` sur `first-example.c`, et explorez l'interface.

   En Frama-C, 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.

   Avec ivette, la fenêtre principale se compose de plusieurs composants.



   — La barre de menu (*menu bar*) à gauche organise les fonctions de haut niveau de l'outil en catégories structurées avec file pour charger et sauvegarder une session frama-c ou help pour l'accès à la documentation de frama-c.

   — La barre d'outils (*tool bar*) en dessous de la barre de menu offre un accès rapide à certaines fonctions fréquemment utilisées, telles que l'exécution d'analyses ou la gestion des sessions.

   — la barre latérale (*side bar*) à gauche permet de naviguer entre les différents fichiers et fonctions analysés, ainsi que d'accéder à des vues spécifiques pour chaque composant. On utilisera principalement la vue “Source” pour naviguer dans le code et la vue “WP goal” pour voir les différentes propriétés à prouver.

   — la zone laboratoire (*laboratory*) au centre affiche le code source normalisé par Frama-C, ainsi que les différentes propriétés à prouver. On peut naviguer entre les différentes fonctions et propriétés en utilisant la barre latérale. Elle consiste en 1 à 4 panneaux, selon les besoins de l'utilisateur. Le

panneau de gauche affiche le code source normalisé par Frama-C, tandis que le panneau de droite affiche le code source original. Les panneaux du bas affichent les différentes propriétés à prouver, ainsi que les résultats des analyses.

2. 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. Notez qu'une coche verte (ou un autre symbole en fonction de votre thème de couleur) apparaît à côté 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.
3. Vous semble-t-il normal que la spécification ait été prouvée ? Pouvez-vous trouver des exemples d'exécutions où une de ces fonctions échouera ? 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.
4. Relancez Frama-C ou ivette avec l'option `-rte` (pouvant être couplée avec les précédentes). Vous noterez que des assertions ont été générées mais qu'elles ne sont pas prouvées, et que les contrats sont maintenant prouvés sous condition que ces nouvelles assertions le soient.

3 Exercice 2

Placez-vous dans le dossier `tp5/Exo2`.

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 ainsi qu'utiliser des opérateurs logiques classiques (conjonction `&&`, disjonction `||`, implication `==>`, ...). La valeur des variables et arguments sera évaluée dans le contexte de retour de la fonction.
On peut décrire les propriétés avec l en utilisant des expressions sur les valeurs des arguments. Par exemple, `@ensures \result == a + b;` est une postcondition disant que la valeur de retour de la fonction doit être égale à la somme des arguments `a` et `b`.

> **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).

4 Exercice 3

Placez-vous dans le dossier `tp5/Exo3`.

> **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. Attention ici, on 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. Ecrivez 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 fichier) 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.

5 Exercice 4

Placez-vous dans le dossier `tp5/Exo4`.

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.

6 Exercice 5

Placez-vous dans le dossier `tp5/Exo5`.

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.

> **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.

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. 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.

> **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 clauses `assumes F;` (où `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 comportements A, B et C sont disjoints).

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.

   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.

7 Exercice 6

Placez-vous dans le dossier `tp5/Exo6`.

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).