

Лекция 12. Язык ACSL. Особенности дедуктивной верификации в системе Frama-C.

Цель лекции

Познакомиться с языком формальной спецификации программ на языке Си. Увидеть особенности формальной спецификации и верификации на уровне языка Си с учетом построения модели программы в системе Frama-C.

Содержание

1 Система Frama-C

2 Язык ACSL

Frama-C

- Система статического анализа Си-программ.
- Состоит из ядра и плагинов. Ядро – это фронтенд анализа, плагин – бекэнд анализа.
- Один из плагинов (AstraVer) – дедуктивная верификация.
- Плагины могут взаимодействовать друг с другом для компенсации недостатков друг друга. Пример: есть плагин, который сам выводит несложные инварианты цикла, и уже на этой основе другой плагин сообщает об ошибке. Всё это делается полностью автоматически.
- `frama-c -av file.c ...`

Работа на уровне исходного кода

- Исходный код может быть снабжен аннотациями для более точного анализа. Для дедуктивной верификации аннотациями записывается спецификация, инварианты цикла и т.п.
- Причем пользователь Frama-C работает исключительно на уровне исходного кода (иначе будет тяжело автоматизировать комбинирование плагинов).
- На уровне исходного кода не доступна его модель на WhyML. Эта модель может иметь существенные особенности по сравнению с исходным Си-кодом, важные для верификации.

Модель кода WhyML не доступна

- (-) Например, нельзя написать лемму, в которой используются функциональные символы из модели памяти.
- (+) Плагин *AstraVer* может самостоятельно строить модель программы, применяя различные оптимизации для более эффективной верификации.

Содержание

1 Система Frama-C

2 Язык ACSL

Что такое ACSL

- ACSL – ANSI/ISO C Specification Language.
- Frama-C использует подмножество языка ACSL.
- Исходный код дополняется *аннотациями*: комментариями
`/*@ ... */`

Аннотации ACSL для спецификации функции

Спецификация функции – это аннотация, которая расположена перед заголовком функции.

- `requires expr;` – предусловие (можно несколько `requires` или ни одного)
- `ensures expr;` – постусловие (можно несколько `ensures` или ни одного)
- `decreases expr;` – оценочная функция для рекурсивной функции (фундированное множество – то же, что в WhyML)
- Синтаксис выражений похож на WhyML, но он сделан более похожим на Си (синтаксис кванторов!).

Типы данных

- `integer` – бесконечный целый тип (только для спецификаций)
- Все Си-шные типы тоже есть
- В операциях с `int` и `integer` происходит преобразование `int` к `integer`

Модель памяти недоступна

- Модель памяти недоступна, поэтому для указателей надо использовать Си-шное разыменование, сложение, вычитание, сравнение.
- Конструкции `\old`, `\at`
- Метки памяти – это Си-метки + Pre, Here, Post
- Не путать `\at(*p,L)` и `*\at(p,L)`

Спецификация фрейма функции

- Модель памяти недоступна, поэтому для фрейма ввели дополнительные части спецификации функции:
- `assigns Locations`; – валидная память за пределами `Locations` не меняет своего значения при выходе из функции
- `allocates Locations`; – валидная память за пределами `Locations` не меняет своего статуса аллоцированности при выходе из функции (если была освобождена, остается освобожденной; была выделенной, остается выделенной); `Locations` вычисляется в состоянии памяти при возврате из функции
- `frees Locations`; – то же, что `allocates`, но `Locations` вычисляется при входе в функцию

Спецификация циклов

- Аннотация цикла записывается перед циклом. Точки сечения, фундированное множество – те же, что в WhyML.
- `loop invariant expr;` – индуктивное утверждение (может быть несколько или отсутствовать)
- `loop variant expr;` – оценочная функция (если ее нет, то она равна 0)
- `loop assigns Locations;` – фрейм цикла

Символы

- Типы-символы, функциональные и предикатные символы, аксиомы пишутся в аннотациях `axiomatic`
- Функциональный символ начинается со слова `logic`
- Предикатный символ начинается со слова `predicate`
- Аксиома начинается со слова `axiom`
- Полиморфные типы не поддерживаются

Снова без модели памяти

- Модель памяти не доступна даже в axiomatic, это нужно учитывать.
- Если нужны метки памяти для функциональных и предикатных символов, их надо писать в фигурных скобках после имени символа, а потом использовать в конструкции `\at`.
- Меток может быть несколько – и это удобно! (для разыменования каждого указателя своя метка – и можно использовать такой предикатный символ в очень разном контексте, не создавая много однотипных предикатных символов)

Аннотации для верификации

- `assert expr;`
- `ghost`-метки и локальные переменные (тип - только Си-шный), `ghost`-блоки (только с Си-шным кодом): это из-за того, что в Frama-C фронтенд кода принимает только Си-код (не `ghost`-аннотация анализируется фронтендом отдельно, поэтому с ней нет этих проблем)
- `lemma` – леммы

Лемма-функции

Пример:

```
/*@ ghost
   /\@ lemma
       requires ....
       ensures ....

   @/
   void lemmafunction (....) {
       ....
   }

*/
```

Построение теорий WhyML и их зависимости

- Каждый axiomatic и глобальная лемма становятся отдельной теорией.
- В теорию импортируются все теории, в которых объявляются символы, используемые в первой теории.
- Для более тонкого управления импортированием надо объявить символ и воспользоваться им в нужном месте.
- В условиях верификации используются все леммы, расположенные до верифицируемой функции.

Триггеры

- В кванторах ACSL нет триггеров.
- Солвер будет выбирать триггер, исходя из квантора, каким он будет в WhyML! Квантор может сильно отличаться от его ACSL-варианта.

Переносимость Си-программ и верификации

- В Си размеры типов выбирает платформа, а не язык.
- Frama-C фиксирует размеры типов во фронтенд анализе (опциями можно настраивать эти размеры). Дедуктивная верификация делается с этими размерами типов.
- Последовательность вычислений тоже фиксируется фронтендом Frama-C.