Лекция 4. Формальная спецификация и верификация Си-программ

Цель лекции

Дать краткое введение в язык формальной спецификации Си-программ ACSL и среду формальной верификации AstraVer.

Содержание

ACSL

- ACSL – ANSI/ISO C Specification Language.
- Исходный код дополняется *аннотациями*: комментариями `/*@ ... */`
- Запуск верификации: `frama-c -av <file.c>`:
  1. Генерируются условия верификации на Why3
  2. Открывается окошко Why3IDE
  3. Пользователь запускает солверы для доказательства условий верификации

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)

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`

Спецификация циклов

- Аннотация цикла записывается перед циклом.
- Точки сечения – перед условием цикла.
- Фундированное множество – неотрицательные целые числа с отношением «меньше».
- `loop invariant expr;` – индуктивное утверждение (может быть несколько или отсутствовать)
- `loop variant expr;` – оценочная функция (если ее нет, то она равна 0)

Предикаты и лоджики

Для повышения читаемости спецификаций можно давать имена выражениям-предикатам и выражениям-не предикатам:

- `predicate positive(integer x) = x >= 0;` – определяет имя `positive` как имя предиката для использования в аннотациях
- `logic integer square(integer x) = x * x;` – определяет имя `square` как имя «функции» для использования в аннотациях
- рекурсивные определения поддерживаются, но они могут давать неожиданный результат
- аннотации используются только для формулирования условий верификации и проведения символьных преобразований над ними в солверах, так что лоджик – это не функция (лишь текстуальная замена)

Указатели: модель памяти

- Память состоит из бесконечного множества блоков.
- Блок имеет фиксированный тип элемента.
- Блок имеет фиксированный размер (количество элементов).
- Блок может быть не аллоцирован или аллоцирован.
- Блоку соответствует бесконечное количество указателей.
- У любого блока (даже не аллоцированного) есть указатель на начало блока («базовый адрес»)
- Указатели типизированные (из блока).

Указатели: спецификация

- `\valid(p)`, `\valid(p + (0 .. n-1))` – валидность указателя `p` и диапазона указателей `p`, ..., `p + (n-1)` (обе границы включаются в диапазон)
- `\offset_min(p)`, `\offset_max(p)` – минимальное и максимальное смещение для указателя `p`, чтобы он был валидным, если блок указателя аллоцирован;
- `\base_addr(p) == \base_addr(q)` – проверка, что `p` и указателя находятся в одном блоке
- `\allocable(p)` – блок указателя `p` аллоцирован
- `\freeable(p)` – блок указателя `p` не аллоцирован и `p` указывает на начало блока

at, old, метки памяти

- `\old(expr)` – выражение `expr` в момент вызова функции (`\old` можно использовать только в постусловии)
- `\at(expr, L)` – выражение `expr` в метке памяти `L`
- Метки памяти – это Си-метки + `Pre`, `Here`, `Post`
- Можно указывать метку и так: `\valid{L}(p)`, `\freeable{L}(p)` и т.п.
- Метки могут быть у `predicate` и `logic`, но написать квантор по меткам нельзя (может быть даже несколько меток!)
- Не путать `\at(*p,L)` и `*\at(p,L)`

Спецификация фрейма 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 аллоцирована, остается такой же; была не аллоцирована, остается такой же); `Locations` вычисляется в метке памяти `Post`
- `frees Locations;` – то же, что `allocates`, но `Locations` вычисляется в метке памяти `Pre`
- Не путать `allocates` и `allocable`, `frees` и `freeable`

Глобальные инварианты

- Это утверждения, которые выполнены всегда, когда программа находится в определенной точке программы
  - строгие - везде
  - слабые - при вызове и при возврате из каждой функции
- Инвариант глобальных переменных:

```
global invariant positive: size >= 0;
```

- Инвариант (каждой переменной) типа:

```
type invariant valid_array(Array *a) =
a->size >= 0 && \valid(a->data + (0 .. a->size - 1));
```

Проблемы глобальных инвариантов

- Пусть у некоторого типа должен быть инвариант. У любой ли функции этот инвариант должен быть выполнен хотя бы в слабом смысле?
- Для функции-конструктора? (а что это в Си?)
- Для функции-деструктора? (а что это в Си?)
- Для вспомогательной функции, вызываемой из «публичной» функции? Любой такой вспомогательной функции?

Подход AstraVer

- Надо точнее специфицировать, когда должен быть выполнен глобальный инвариант. Вводить дополнительные конструкции в язык спецификации?
- В AstraVer глобальные инварианты не вставляются автоматически в спецификации функций!
- Это просто предикаты, которые надо явно указать в спецификациях нужных функций.

И это не все проблемы

- Пусть некоторый тип является частью другого типа. Причем не все корректные значения внутреннего типа являются допустимыми в рамках внешнего типа.
- Каков должен быть инвариант внутреннего типа? Как доказать, что модификация переменной внутреннего типа не нарушает инварианта переменной внешнего типа, если нет доступа до переменной внешнего типа?
- До сих пор нет лучшего ответа на этот вопрос.

Содержание

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.

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.

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.