Frama-C and ACSL

Michael Wipplinger

December 2, 2019

- What is Frama-C?
- Installation
- How to use Frama-C
- Dynamic Verification
- Static Verification
- ACSL Basics
  - Function Contracts
  - Predicates
  - Loop Invariants
- Conclusion
- Sources

What is Frama-C?

Frama-C is a platform for the analysis of C code. It provides:

- a "normalization" of your code
- value analysis
- metrics of functions and programs
- slicing of code
- impact analyis
- prove methods for specifications written in ACSL

Installation

- Install a WSL (Windows only)
- Install XLaunch Server (Windows only)
- Install opam
- With opam install Frama-C

Precise instructions and downloads can be found HERE.

How to use Frama-C

- Write your code with ACSL specifications in the comments using your favourite text editor
- Compile your code as always
- Launch Frama-C and open the source files (console: frama-c-gui filename.c)
- analyse
- In order to edit your code, run a text editor simultaneously, save your changes in the editor and click the "Refresh Button" in the Frama-C GUI (you might need to re-compile)
- If you cannot open a file, it might be due to a syntax error (try using the console command to get the location and type of error)

Dynamic Verification

Dynamic Verification is performed during runtime. For example:

- E-ACSL
  - Builds a program that does the same things but reports an error every time a ACSL specification is violated during runtime.
- StaDy
  - Does the same as E-ACSL but searches the argument space to generate counterexamples.

Static Verification

Static Verification relies only upon source code analysis.
Expamples are:

- value analysis
  - Shows possible or exact values for variables.
- slicing
  - Splits a program into smaller, simpler programs
- Provers
  - Provers like Alt-Ergo and Coq try to prove properties defined in ACSL

ACSL Basics

ACSL stands for ANSI C Specification Language and is a formal language that allows us to specify properties of functions and variables which can than be interpreted by different applications. We can, for example, define:

- assertions, can be placed everywhere in the code and describe properties that should hold at that point in the program
- requirements for function arguments and what is supposed to hold for the result (function contracts).
- predicates (as in first order logic)
- axioms and lemmas (e.g. for algebraic data types or to help the prover)

ACSL Basics

- ACSL code is written in comments of the form `/*@ ... */` or `//@ ...`.
- Expressions are formed with standard C operators and types as well as Built-in constructs like
    - `\forall` and `\exists`
    - `\true` and `\false`
    - `==>` and `<==>`
    - mathematical integers and reals
    - `\at(term,label-id)`
    - `\valid(ptr)` and other predefined predicates

Function Contracts

For any function we can define specifications, documenting what the function does. The syntax is as follows:

```
/*@
reqires predicate;*
terminates predicate;
decreases term;
assigns location (, location)* | \nothing;
ensures predicate;
behavior bahavior_name:*
  assumes predicate;*
  requires predicate;*
  assigns location (, location)* | \nothing;
  ensures predicate;*
```

```
complete behaviors behavior_name (, behavior_name)*;*
disjoint behaviors behavior_name (, behavior_name)*;*
*/
type function_name(...)
{
...
}
```

Where * implies that the prior expression can be repeated and | means one can choose between the left and right option.

Predicates

Can be defined directly:

```
/*@
predicate predicate_name{State}(arguments) = expression;
*/
e.g.
/*@
predicate divides(int a, int b) = {\exists integer c; c*a =
*/
```

Predicates

Can be defined inductively, e.g.:

```
/*@
inductive is_gcd(int g, int a, int b) {
 case a_is_zero:
   \forall integer a, integer b; a == 0 ==> is_gcd(b,a,b);
 case b_is_zero:
   \forall integer a, integer b; b == 0 ==> is_gcd(a,a,b);
 case valid_transform:
   \forall integer a, integer b, integer g; is_gcd(g,a,b) &
*/
```

Loop Invariants

A loop invariant is defined right above a C-loop like while, for or do ... while.

```
/*@
loop invariant predicate;*
loop assigns location (, location)*;
for behavior_name:*
  loop invariant predicate;*
  loop assigns predicate;*
  loop variant term;*
loop variant term;*
*/
```

Note that `behavior_name` comes from the behavior defined in the function contract.

Conclusion

- Installation worked well
- Working with Frama-C requires some time to get used to
- Proving properties requires extensive use of ACSL specifications, in a scale similar to writing the program a second time.
- Many libraries are not yet supported.
- There exists good documentation out there, but it is not as easy to find as for other languages
- However the basics of ACSL are very intuitive and if it works, it works for sure.

- Links:
    - Official Mini-Tutorial by Virgile Prevosto here
    - ACSL by Example by Jochen Burghardt et. al. here
    - Official Documentation of ACSL in Frama-C by Patrick Baudin et. al. here
    - Some very useful examples for beginners on github here