Specification and Proof of Programs with Frama-C

Nikolai Kosmatov

firstname.lastname@cea.fr

CEA LIST

CentraleSupélec, January 16, 2018

Motivation

Main objective:

Rigorous, mathematical proof of semantic properties of a program

- functional properties
- safety:
  - all memory accesses are valid,
  - no arithmetic overflow,
  - no division by zero, ...
- termination
- ...

Our goal

In this lesson, we will see

- how (Floyd-)Hoare logic is used for proof of programs
- how to specify a C program
- how to prove it with an automatic tool
- how to understand and fix proof failures

Outline

Outline

Frama-C: A brief history

- 90's: CAVEAT, a Hoare logic-based tool for C programs
- 2000's: CAVEAT used by Airbus during certification of the A380
- 2002: Why tool and its C front-end Caduceus
- 2006: Joint project to write a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2009: Hoare-logic based Frama-C plugin Jessie developed at INRIA
- 2012: New Hoare-logic based plugin WP developed at CEA LIST
- Frama-C today:
  - Most recent release: Frama-C Sulfur (v.16)
  - Multiple projects around the platform
  - A growing community of users

Frama-C at a glance

- FRAmework for Modular Analysis of C programs
  - Various plugins: CFG, value analysis (abstract interpretation), impact analysis, dependency analysis, slicing, program proof, . . .
- Developed at CEA LIST and INRIA Saclay (Proval/Toccata team)
- Released under LGPL license
- Kernel based on CIL library [Necula et al. – Berkeley]
- Includes ACSL specification language
- Extensible platform
  - Adding specialized plugins is easy
  - Collaboration of analyses over the same code
  - Inter-plugin communication through ACSL formulas
- `http://frama-c.com/`

ACSL: ANSI/ISO C Specification Language

Presentation

- Based on the notion of contract, like in Eiffel
- Allows the users to specify functional properties of their programs
- Allows communication between various plugins
- Independent from a particular analysis
- ACSL manual at `http://frama-c.com/acsl.html`

Basic Components

- First-order logic
- Pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-ins predicates and logic functions, particularly over pointers: `\valid(p) \valid(p+0..2), \separated(p+0..2,q+0..5), \block_length(p)`

WP plugin

- Hoare-logic based plugin, developed at INRIA Saclay
- Proof of functional properties of the program
- Modular verification (function per function)
- Input: a program and a specification in ACSL
- WP generates verification conditions (VCs)
- Use of Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Simplify, Z3, Yices, CVC3, ...
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?

WP plugin

- Hoare-logic based plugin, developed at INRIA Saclay
- Proof of functional properties of the program
- Modular verification (function per function)
- Input: a program and a specification in ACSL
- WP generates verification conditions (VCs)
- Use of Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Simplify, Z3, Yices, CVC3, . . .
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?
  - If the specification is wrong, the program can be wrong
- Limitations
  - Casts between pointers and integers
  - Limited support for union type
  - Aliasing requires some care. . .

In this lesson

In this lesson we use

- Frama-C
- WP plug-in
- Alt-Ergo

To run Frama-C/WP on a C program `file.c`

- `frama-c -wp file.c`

To run Frama-C/WP also for assertions preventing runtime errors

- `frama-c -wp -wp-rte file.c`

Outline

Contracts

- Goal: specification of imperative functions
- Approach: give assertions (i.e. properties) about the functions
  - Precondition is supposed to be true on entry (ensured by callers of the function)
  - Postcondition must be true on exit (ensured by the function if it terminates)
- Nothing is guaranteed when the precondition is not satisfied
- Termination may or may not be guaranteed (total or partial correctness)

Primary role of contracts

- Main input of the verification process
- Must reflect the informal specification
- Should not be modified just to suit the verification tasks

What is Hoare Logic?

- A formal system allowing to reason about programs (by Floyd 1967, Hoare 1969,. . .)
- Main idea: provided some property is satisfied before some command, show another property after the execution of the command
- A Hoare triple: $\{P\}\ C\ \{Q\}$ where P, Q are assertions and C is a command (a piece of program)
- Assertions are first-order formulae over program variables
- The Hoare triple $\{P\}\ C\ \{Q\}$ is valid if
  - whenever P is true before C and C terminates, Q is true after C
- Termination is not guaranteed (partial correctness)
  - If necessary, it may be proved separately (total correctness, cf below)
- Ex.: $\{30 = z\}\ x := 10;\ y := 20\ \{x + y = z\}$ is valid
- Ex.: $\{30 = z\}\ x := 10;\ y := 20;\ z := t\ \{x + y = z\}$ is not valid

Hoare logic rules

- A set of inference rules of the form

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_k}{Q}$$

- Means: if premises $P_1, \ldots, P_k$ are all true, then conclusion Q is true
- A rule without premises is an axiom
- A proof in Hoare logic deduces valid Hoare triples using inference rules, for example:

$$\frac{\dfrac{R \qquad \dfrac{}{\{1+2 \leq z\}\, x := 1\, \{x+2 \leq z\}}}{\{3 = z\}\, x := 1\, \{x+2 \leq z\}} \qquad \dfrac{}{\{x+2 \leq z\}\, y := 2\, \{x+y \leq z\}}}{\{3 = z\}\, x := 1;\ y := 2\, \{x+y \leq z\}}$$

where R is $3 = z \;\Rightarrow\; 3 \leq z$

The rules

$$\text{skip}\ \frac{}{\{Q\}\ \mathit{skip}\ \{Q\}}$$

$$\text{ass}\ \frac{}{\{Q[x \leftarrow e]\}\ x := e\ \{Q\}}$$

$$\text{seq}\ \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1;\ C_2\ \{R\}}$$

$$\text{if}\ \frac{\{P \wedge e\}\ C_1\ \{Q\} \qquad \{P \wedge \neg e\}\ C_2\ \{Q\}}{\{P\}\ \mathit{if}\ \ e\ \ \mathit{then}\ \ C_1\ \ \mathit{else}\ \ C_2\ \{Q\}}$$

$$\text{while}\ \frac{\{I \wedge e\}\ C\ \{I\}}{\{I\}\ \mathit{while}\ \ e\ \ \mathit{do}\ \ C\ \{I \wedge \neg e\}}$$

$$\text{cons}\ \frac{P \Rightarrow P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

$Q[x \leftarrow e]$ is the assertion Q where all free occurences of x are replaced by e

More proof examples

Identify the rules being applied, and complete:

- $\{true\}\ if\ \ x > 10\ \ then\ \ y := 10\ \ else\ \ y := x\,\{y \le 10\}$

$$\dfrac{\dfrac{x > 10\ \Rightarrow\ 10 \le 10 \qquad \dfrac{}{\{10 \le 10\}\, y := 10\,\{y \le 10\}}}{\{x > 10\}\, y := 10\,\{y \le 10\}} \qquad \dfrac{}{\{x \le 10\}\, y := x\,\{y \le 10\}}}{\{true\}\ if\ \ x > 10\ \ then\ \ y := 10\ \ else\ \ y := x\,\{y \le 10\}}$$

- $\{x \ge 0\}\, n := 0;\ \ while\ \ n < x\ \ do\ \ n := n + 1\,\{n = x\}$

$$\dfrac{\dfrac{\vdots}{\{0 \le x\}\, n := 0\,\{0 \le n \le x\}} \qquad \dfrac{\dfrac{\vdots}{\{0 \le n \le x \wedge n < x\}\, n := n + 1\,\{0 \le n \le x\}}}{\{0 \le n \le x\}\ while\ n < x\ do\ n := n + 1\,\{0 \le n \le x \wedge n \ge x\}}}{\{0 \le x\}\, n := 0;\ \ while\ \ n < x\ \ do\ \ n := n + 1\,\{n = x\}}$$

Outline

Safety warnings: arithmetic overflows

Absence of arithmetic overflows can be important to check

- A sad example: crash of Ariane 5 in 1996
- WP can automatically generate VCs to check absence of overflows (option `-wp-rte`)
- They ensure that arithmetic operations do not overflow
- If not proved, an overflow may occur. Is it intended?

Safety warnings: invalid memory accesses

An invalid pointer or array access may result in a segmentation fault or memory corruption.

- WP can automatically generate VCs to check memory access validity (option `-wp-rte`)
- They ensure that each pointer (array) access has a valid offset (index)
- If the function assumes that an input pointer is valid, it must be stated in its precondition, e.g.
  - `\valid(p)` for one pointer `p`
  - `\valid(p+0..2)` for a range of offsets `p, p+1, p+2`

Frame rule

The clause `assigns v1, v2, ... , vN;`

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- No need to specify local variable modifications in the postcondition
  - a function is allowed to change local variables
  - a postcondition cannot talk about them anyway, they do not exist after the function call
- Avoids to state that for any unchanged global variable `v`, we have `ensures \old(v) == v`
- Avoids to forget one of them: explicit permission is required
- If nothing can be modified, specify `assigns \nothing`

Behaviors

Specification by cases

- Global precondition (`requires`) applies to all cases
- Global postcondition (`ensures`, `assigns`) applies to all cases
- Behaviors define contracts (refine global contract) in particular cases
- For each case (each `behavior`)
  - the subdomain is defined by `assumes` clause
  - the behavior's precondition is defined by `requires` clauses
    - it is supposed to be true whenever `assumes` condition is true
  - the behavior's postcondition is defined by `ensures`, `assigns` clauses
    - it must be ensured whenever `assumes` condition is true
- `complete behaviors` states that given behaviors cover all cases
- `disjoint behaviors` states that given behaviors do not overlap

Contracts and function calls

Function calls are handled as follows:

- Suppose function `g` contains a call to a function `f`
- Suppose we try to prove the caller `g`
- Before the call to `f` in `g`, the precondition of `f` must be ensured by `g`
  - VCs is generated to prove that the precondition of `f` is respected
- After the call to `f` in `g`, the postcondition of `f` is supposed to be true
  - the postcondition of `f` is assumed in the proof below
  - modular verification: the code of `f` is not checked at this point
  - only a contract and a declaration of the callee `f` are required

Pre/post of the caller and of the callee have dual roles in the caller's proof

- Pre of the caller is supposed, Post of the caller must be ensured
- Pre of the callee must be ensured, Post of the callee is supposed

Outline

Loops and automatic proof

- What is the issue with loops? Unknown, variable number of iterations
- The only possible way to handle loops: proof by induction
- Induction needs a suitable inductive property, that is proved to be
  - satisfied just before the loop, and
  - satisfied after $k+1$ iterations whenever it is satisfied after $k \geq 0$ iterations
- Such inductive property is called loop invariant
- The verification conditions for a loop invariant include two parts
  - loop invariant initially holds
  - loop invariant is preserved by any iteration

Loop invariants - some hints

How to find a suitable loop invariant? Consider two aspects:

- identify variables modified in the loop
  - variable number of iterations prevents from deducing their values (relationships with other variables)
  - define their possible value intervals (relationships) after k iterations
  - use `loop assigns` clause to list variables that (might) have been assigned so far after k iterations
- identify realized actions, or properties already ensured by the loop
  - what part of the job already realized after k iterations?
  - what part of the expected loop results already ensured after k iterations?
  - why the next iteration can proceed as it does? . . .

A stronger property on each iteration may be required to prove the final result of the loop

Some experience may be necessary to find appropriate loop invariants

Loop invariants - more hints

Remember: a loop invariant must be true

- before (the first iteration of) the loop, even if no iteration is possible
- after any complete iteration even if no more iterations are possible
- in other words, any time before the loop condition check

In particular, a `for` loop

```
for(i=0; i<n; i++) { /* body */ }
```

should be seen as

```
i=0;          // action before the first iteration
while( i<n ) // an iteration starts by the condition check
  {
    /* body */
    i++;      // last action in an iteration
  }
```

Loop termination

- Program termination is undecidable
- A tool cannot deduce neither the exact number of iterations, nor even an upper bound
- If an upper bound is given, a tool can check it by induction
- An upper bound on the number of remaining loop iterations is the key idea behind the loop variant

Terminology

- Partial correctness: if the function terminates, it respects its specification
- Total correctness: the function terminates, and it respects its specification

Loop variants - some hints

- Unlike an invariant, a loop variant is an integer expression, not a predicate
- Loop variant is not unique: if V works, $V + 1$ works as well
- No need to find a precise bound, any working loop variant is OK
- To find a variant, look at the loop condition
  - For the loop `while(exp1 > exp2 )`, try `loop variant exp1-exp2;`
- In more complex cases: ask yourself why the loop terminates, and try to give an integer upper bound on the number of remaining loop iterations

\forall and \exists - hints and examples

- Do not confuse `&&` and `==>` inside `\forall` and `\exists`
- Some common patterns:
  - `\forall integer j; 0 <= j && j < n ==> t[j] == 0;`
  - `\exists integer j; 0 <= j && j < n && t[j] != 0;`
  - Each one here is negation of the other
- A shorter form:
  - `\forall integer j; 0 <= j < n ==> t[j] == 0;`
  - `\exists integer j; 0 <= j < n && t[j] != 0;`
- With several variables:
  - `\forall integer i,j; 0 <= i <= j < length ==> a[i]<=a[j];`
  - `\exists integer i,j; 0 <= i <= j < length && a[i]>a[j]`

Referring to another state

- Specification may require values at differents program points
- Use `\at(e,L)` to refer to the value of expression `e` at label `L`
- Some predefined labels:
  - `\at(e,Here)` refers to the current state
  - `\at(e,Old)` refers to the pre-state
  - `\at(e,Post)` refers to the post-state
- `\old(e)` is equivalent to `\at(e,Old)`

Outline

Proof failures

A proof of a VC for some annotation can fail for various reasons:

- incorrect implementation (→ check your code)
- incorrect annotation (→ check your spec)
- missing or erroneous (previous) annotation (→ check your spec)
- insufficient timeout (→ try longer timeout)
- complex property that automatic provers cannot handle.

Analysis of proof failures

When a proof failure is due to the specification, the erroneous annotation may be not obvious to find. For example:

- proof of a "loop invariant preserved" may fail in case of
  - incorrect loop invariant
  - incorrect loop invariant in a previous, or inner, or outer loop
  - missing `assumes` or `loop assumes` clause
  - too weak precondition
  - ...
- proof of a postcondition may fail in case of
  - incorrect loop invariant (too weak, too strong, or inappropriate)
  - missing `assumes` or `loop assumes` clause
  - inappropriate postcondition in a called function
  - too weak precondition
  - ...

Analysis of proof failures (Continued)

- Additional statements (`assert`, `lemma`, ...) may help the prover
  - They can be provable by the same (or another) prover or checked elsewhere
- Separating independent properties (e.g. in separate, non disjoint behaviors) may help
  - The prover may get lost with a bigger set of hypotheses (some of which are irrelevant)

When nothing else helps to finish the proof:

- an interactive proof assistant can be used
- Coq, Isabelle, PVS, are not that scary: we may need only a small portion of the underlying theory

Outline

Conclusion

- We learned how to specify and prove a C program with Frama-C
- Hoare-logic based tools provide a powerful way to formally verify programs
- The program is proved with respect to the given specification, so
  - Absence of proof failures is not sufficient
  - The specification must be correct
- The proof is automatic, but analysis of proof failures is manual
- Proof failures help to complete the specification or find bugs
- Interactive proof tools may be necessary to finish the proof for complex properties that cannot be proved automatically