Automated Proofs in Frama-C

Michael Wipplinger

February 3, 2020

Revision

- ACSL code is written in comments of the form `/*@ ... */` or `//@ ...`
- We can define predicates, logic functions, lemmas and axioms
- We can use those to define function contracts, that specify properties of the function which can later be verified.

Hoare-Triples

Consider a simple function contract:

```
/*@
requires valid(a) && valid(b);
ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b)
{
int b_save = *b;
*b = *a;
*a = b_save;
}
```

Where we have a pre and a post condition as well as statements to be executed.

Hoare-Triples

We can break this (and every other) contract down to assertions before and after the statements:

```
int swap(int* a, int* b)
{
\\@ assert valid(a) && valid(b);

int b_save = *b;
*b = *a;
*a = b_save;

\\@ assert *a == \old(*b) && *b == \old(*a);
}
```

This denotes a so called **Hoare-Triple** `{P} stmt {Q}`
"if *P* holds, than after running *stmt*, *Q* holds"

Weakest Precondition Calculus

The goal of weakest precondition calculus is to find a condition $p = wp(stmt, Q)$ that $\{p\}\ stmt\ \{Q\}$ holds, and that $P \implies p$ for all P that are a sufficient precondition.

We will take a look at the rules used to derive this precondition for an imperative language without pointers and loops.

Weakest Precondition Calculus Rules

- **Special statements**:
  - $\{Q\}\ skip\ \{Q\}$ (skip command does not do anything)
  - $\{P\}abort\{Q\}$ is true for all P and Q
- **Scalar Assignments**: $\{Q[e/x]\}x := e\{Q\}$
  ($Q[e/x]$ the predicate where all free occurrences of x in the definition of Q are replaced by e)
- **Sequence Rule**: $\dfrac{\exists R_1, R_2\colon \{P\}c_1\{R_1\},\ R_1 \implies R_2,\ \{R_2\}c_2\{Q\}}{\{P\}c_1;c_2\{Q\}}$
- **Conditional**: $\dfrac{\{P \wedge b\}\ c_1\ \{Q\},\ \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ if\ \ b\ then\ c_1\ else\ c_2\ \{Q\}}$
- **Loop Rule**: $\dfrac{\exists I\colon P \implies I,\ \{I \wedge b\}\ c\ \{I\},\ (I \wedge \neg b) \implies Q}{\{P\}\ while\ b\ do\ c\ \{Q\}}$

Proofs

A rough sketch of how a proof is performed:

- Source code is imported into Frama-C (Basic simplification and minor rewrites)
- The WP-Plugin included in Frama-C breaks down function contracts to Hoare-Triples.
- The WP-Plugin computes the weakest precondition for all Hoare-Triples
- For every Hoare-Triple the weakest precondition and the given precondition are passed on to an external prover to show that the given precondition implies the weakest precondition.

Hoare Memory Model

- Does not feature pointers in any way!
- Each variable is represented by (several) logic variables that are passed to the external prover.
- e.g. a variable x will be translated to x_0 at the beginning of the function, x_1 after the first command and so on ...

The Hoare-Model **is** implemented in Frama-C, but it will complain if there are pointers being dereferenced in the program, meaning it is not suited for most programs.

Pointer Memory Models

There are many ways to model the bit-stream that is the accessing and writing values in the heap. A common structure is as follows:

- **Pointer Types**: P, a tuple of an address and the size of the object stored there
- **Heap Variables**: the values $m_1, m_2, \ldots m_k = \bar{m}$ stored in the heap.
- **Read Operation**: $read_T(\bar{m}, p) \mapsto term$
- **Write Relation**: $write_T(\bar{m}, p, v, \bar{m}')$ is true iff writing value v in address p of $\bar{m}$ results in $\bar{m}'$

Pointer Memory Models

Consider the statement `(*p)++;`. In the formal model this would mean:

- $A_p = read_{int*}(\bar{m}, P)$, $(P = adressofp)$
- $V_p = read_{int}(\bar{m}, A_p)$
- $write_{int}(\bar{m}, A_p, V_p + 1, \bar{m}')$

You can see that variables of every type (including pointers) are stored in the heap variable. So any C-variable will just be identified as an address, the value is stored in the heap variable $\bar{m}$.

Hoare-Variables mixed with Pointers

- Variables whose addresses are not used (most pointers) are Hoare-Variables
- Values that are accessed via pointer use a pointer memory model as discussed before
- Very efficient in practice, standard model for WP in Frama-C

Our example from before `(*p)++;` would look like this:

- $V_p = read_{int}(\bar{m}, P)$
- $write_{int}(\bar{m}, P, V_p + 1, \bar{m}')$

Where P is a Hoare-Variable, the address stored in p.

Other Models

- Typed Model (separate $\bar{m}$'s for each data type, formerly the standard model)
- Caveat Model (typed model with some tweaks to be more efficient, not so safe however)
- Bytes Model (one to one simulation of heap. Not implemented!)

Integer Models

- **Machine Integer Model**: Overflowing is allowed by default, but can be disabled. If overflowing is allowed, the addition is interpreted as the mathematical operation on unbound integers and a appropiate modulo (depends on long/short and signed/unsigned).
- **Natural Integers**: unbounded mathematical integers as we know them. Translation to Machine Integers works by modulo again.

Reals / Float Models

- **Float Model**: based upon IEEE 754 specifications for floating-point numbers. However it provides little support for proving properties with automated provers.
- **Reals**: floating-point operations are "transformed" on reals, with no rounding. This is completely unsound with respect to C and IEEE semantics. Properties proved with this model can not be recovered for floating-point numbers.

Simplification

The following simplifications are (by default) performed by the Frama-C WP-Plugin:

- **Logic**: Formulae are normalized by commutativity, associativity, absorption and neutral elements. The Qed engine also provides some simplifications from sequent calculus.
- **Arithmetic**: Terms and (in)equalities are simplified by commutativity, associativity, absorption and neutral elements and even linear factorization.
- **Arrays**: elimination of consecutive accesses/updates

SMT Solvers

- SMT stands for **Satisfiability modulo Theories** and is a decision problem (true/false)
- A theory is a set of first order logic formulae
- Most are based on Boolean-SAT solvers
- Most SMT-Solvers come with implemented theories such as:
  - empty theory
  - linear/non-linear integer arithmetic
  - floating point arithmetic
  - theories regarding data types and arrays

Available in Frama-C:

- Alt-Ergo (standard)
- Beagle
- CVC3 / CVC4

Proof Assistants

- Also called interactive provers
- They do not generate a proof automatically but mechanically check whether a given proof is correct
- They also rely upon libraries with predefined theories

Available in Frama-C:

- Coq
- Isabelle/HOL
- PVS

Axioms, Lemmas and Predicates

- The axioms, lemmas and predicates we defined essentially work as an expansion of the theories the provers already know.
- Since the automatic (SMT) solvers are not complete, one might need to help out, providing lemmas
- To verify the lemmas we can use the interactive provers

The Problem with Total Verification

- Even though SMT-Solvers became much more powerful over the last years, they still require help in form of lemmas, even for "easy" proofs.
- Often one even has to implement several (slightly different) instances of the same lemma..
- The result is, that the verification engineer is forced to work with both the in-code specifications and external proof assistants - two complex systems using a different language.

The Auto-Active Approach

- Originally auto-active verification describes an approach where the user input is supplied before the generation of the verification conditions.
- This can be achieved using ghost code, so called "lemma-functions" and other features of ACSL.

Figure: A 2019 study by Blanchard et al. that focused on auto-active verification led to the following results:

	Lemmas, incl. lemma functions & lemma macros	Generated goals	Goals proved with Coq	Lines of code		Execution time
				Lemmas, incl. l.fun./macros	Guiding annotations	
Case study (1). The memory management module MEMB (70 lines of C code)						
Classic	15	134	15	33	20	47 s
Auto-active	3	217	1	25	25	19 s
Case study (2). The linked list module (176 lines of C code)						
Classic	24	805	19	163	708	24 min
Auto-active	17	1631	1	366	629	21 min
Case study (3). *ACSL by Example*, v. 17.2.0 (630 lines of C code)						
Classic	87	1398	40	594	485	92 min
Auto-active	53	1790	0	670	611	78 min

Sources

- Frama-C WP models: Frama-C WP Manual
- Weakest Precondition/Hoare Calculus: Hoare Calculus and Predicate Transformers
- SMT-Solvers: Wikipedia
- Auto-Active Verification: Lemma Functions for Frama-C:C Programs as Proofs
- Auto-Active Verification: Towards Full Proof Automation in Frama-C