Frama-C Training Session
Introduction to ACSL and its GUI

Virgile Prevosto

CEA List

October 21st, 2010
Presentation

ACSL Specifications
  Function contracts
  First-order logic
  Loops
  Assertions

Deductive Verification
  Hoare logic
  Pointers and Memory
  Jessie Plugin
Presentation

ACSL Specifications
  Function contracts
  First-order logic
  Loops
  Assertions

Deductive Verification
  Hoare logic
  Pointers and Memory
  Jessie Plugin
Main objective

Statically determine some semantic properties of a program

- safety: pointer are all valid, no arithmetic overflow, ...
- termination
- functional properties
- dead code
- ...

Embedded code

- Much simpler than desktop applications
- Some parts are critical, i.e. a bug have severe consequences (financial loss, or even dead people)
- Thus a good target for static analysis
Polyspace Verifier  Checks for (absence of) run-time error C/C++/Ada
http://www.mathworks.com/products/polyspace/

ASTRÉE  Absence of error without false alarm in SCADE-generated code
http://www.di.ens.fr/~cousot/projets/ASTREE/

Coverity  Checks for various code defects (C/C++/Java)
http://www.coverity.com
a3  Worst-case execution time and Stack depth
http://www.absint.com/

FLUCTUAT  Accuracy of floating-point computations and origin of rounding errors
http://www-list.cea.fr/labos/fr/LSL/fluctuat/

Frama-C  A toolbox for analysis of C programs
http://frama-c.com/
Presentation

A brief history

- 90’s: CAVEAT, an Hoare logic-based tool for C programs
- 2000’s: CAVEAT used by Airbus during certification process of the A380
- 2002: Why 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)
- today:
  - Frama-C Boron
  - Multiple projects around the platform
  - A growing community of users
A modular architecture

Kernel:
- CIL (U. Berkeley) library for the C front-end
- ACSL front-end
- Global management of analyzer’s state

Various plug-ins for the analysis
- Value analysis (abstract interpretation)
- Jessie (translation to Why)
- Slicing
- Impact analysis
- ...
ACSL: ANSI/ISO C Specification Language

Presentation

- Based on the notion of contract, à la Eiffel
- Allow the users to specify functional properties of their programs
- Allow communication between the various plugin
- Independent from a particular analysis

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: \texttt{valid(p)}, \texttt{valid(p+0..2)}, \texttt{separated(p+0..2,q+0..5)}, \texttt{block_length(p)},...
Presentation

ACSL Specifications
Function contracts
First-order logic
Loops
Assertions

Deductive Verification
Hoare logic
Pointers and Memory
Jessie Plugin
ACSL Specifications - Function contracts

Key Ingredients

Specification of a function
- Contract between caller and callee
- Callee requires some pre-conditions from the caller
- Callee ensures some post-conditions hold when it returns

A first example

```c
unsigned int M;
/*@
   requires @valid(p) && @valid(q);
   ensures M == (*p + *q) / 2;
*/
void mean(unsigned int* p, unsigned int* q) {
   if (*p >= *q) { M = (*p - *q) / 2 + *q; }
   else { M = (*q - *p) / 2 + *p; }
}
```
ACSL Specifications - Function contracts

Specification of Side Effects

The specification
/*@ 
  requires \valid(p) && \valid(q);
  ensures M == (*p + *q) / 2;
*/

void mean(unsigned int* p, unsigned int* q);

A valid implementation
The specification

```c
/*@ 
requires \valid(p) && \valid(q);
ensures M == (*p + *q) / 2;
*/

void mean(unsigned int* p, unsigned int* q);
```

A valid implementation

```c
void mean(int *p, int* q)
{
    *p = *q = M = 0;
}
```
The specification

/*@

requires \valid(p) && \valid(q);
ensures M == (*p + *q) / 2;
ensures *p == \old(*p) && *q == \old(*q);
*/

void mean(unsigned int* p, unsigned int* q);

A valid implementation
Specification of Side Effects

The specification

`/*@`

`requires \valid(p) \&\& \valid(q);`
`ensures M == (*p + *q) / 2;`
`ensures *p == \old(*p) \&\& *q == \old(*q);`
`*/`
`void mean(unsigned int* p, unsigned int* q);`

A valid implementation

```c
int A = 42;
void mean(int *p, int* q) {
    if (*p >= *q) ... else ...
    A = 0;
}
```
The specification
/*@
   requires \valid(p) && \valid(q);
   ensures M == (*p + *q) / 2;
   assigns M;
*/
void mean(unsigned int* p, unsigned int* q);

A valid implementation
The specification

```c
/*@ 
requires \valid(p) && \valid(q);
ensures M == (*p + *q) / 2;
assigns M;
*/
void mean(unsigned int* p, unsigned int* q);

A valid implementation

```c
void mean(int *p, int* q) {
    if (*p >= *q) { M = (*p - *q) / 2 + *q; }
    else { M = (*q - *p) / 2 + *p; }
}
Informal spec

- Input: a sorted array and its length, an element to search.
- Output: index of the element or -1 if not found

Towards a formal specification

```c
int find_array(int* arr, int length, int query);
```

- How to specify the two distinct outcome?
- What does that mean for arr to be sorted?
- How to prove the implementation?

Deductive Verification
/*@ behavior found:
  assumes \exists integer i;
  0\leq i< length && arr[i] == query;
  ensures 0\leq \result< length &&
          arr[\result] == query;

behavior not_found:
  assumes \forall integer i;
  0\leq i< length => arr[i] != query;
  ensures \result == -1;

complete behaviors; disjoint behaviors;
*/

int find_array(int* arr, int length, int query);
Predicate definition

/*@ */

predicate sorted{L}(int* arr, int length) =
\forall integer i,j;
0<=i<=j<length ==> arr[i] <= arr[j];
*/

/*@ requires sorted{Here}(arr,length);
requires \valid(arr+(0..length-1));
requires length >= 0; */

int find_array(int* arr, int length, int query);
/*@ inductive sorted{L}(int* arr, int length) { 
    case singleton{L}:
        \forall int* arr; sorted{L}(arr,0);
    case trans{L}:
        \forall int* arr, integer length;
        sorted{L}(arr,length)
        && arr[length-1] <= arr[length]
        ==> sorted{L}(arr,length + 1);
} */

/*@ requires sorted{Here}(arr,length);
requires \valid(arr+(0..length-1));
requires length >= 0; */
int find_array(int* arr, int length, int query);
int find_array(int* arr, int length, int query)
{
    int min = 0;
    int max = length - 1;
    int mean;
    while (min <= max) {
        mean = min + (max - min) / 2;
        if (arr[mean] == query) return mean;
        if (arr[mean] < query) min = mean + 1;
        else max = mean - 1;
    }
    return -1;
}
Loop annotations

/*@ loop invariant 0 <= min < length;
loop invariant 0 <= max < length;
loop invariant
\forall integer i;
  0 <= i < min == > arr[i] < query;
loop invariant
\forall integer i;
  max < i < length == > arr[i] > query;
loop assigns mean, min, max;
loop variant max - min;
*/

while (min <= max) {
  ...
}
while (min <= max) {
    mean = min + (max - min) / 2;
    //@ assert min <= mean <= max; */
    if (arr[mean] == query) return mean;
    if (arr[mean] < query)
        min = mean + 1;
    else
        max = mean - 1;
Presentation

ACSL Specifications
  Function contracts
  First-order logic
  Loops
  Assertions

Deductive Verification
  Hoare logic
  Pointers and Memory
  Jessie Plugin
Deductive Verification - Hoare logic

- Introduced by Floyd and Hoare (70s)
- Hoare triple: $\{P\} s \{Q\}$, meaning: *If $P$ holds, then $Q$ will hold after the execution of statement $s$*
- Deduction rules on Hoare triples: *Axiomatic semantic*
Deductive Verification - Hoare logic

Some rule examples

\[
\begin{align*}
\{ P \} \{ \top \} & \quad \vdash \quad \{ P' \} \{ Q \} \quad \text{if } P \Rightarrow P' \quad \text{and } Q' \Rightarrow Q \\
\end{align*}
\]

\[
\begin{align*}
\{ P \} s_1 \{ R \} & \quad \vdash \quad \{ R \} s_2 \{ Q \} \quad \text{if } \{ P \} \{ \top \} \quad \text{and } e \text{ evaluates without error} \\
\end{align*}
\]

\[
\begin{align*}
\{ P \land e \} s_1 \{ Q \} & \quad \vdash \quad \{ P \land \neg e \} s_2 \{ Q \} \\
\end{align*}
\]

\[
\begin{align*}
\{ I \land e \} s_1 \{ I \} & \quad \vdash \quad \{ I \land \neg e \} s_2 \{ I \} \\
\end{align*}
\]

\[
\begin{align*}
\{ I \} \text{while } (e) & \quad \vdash \quad \{ I \land \neg e \} s \{ I \land \neg e \} \\
\end{align*}
\]
Weakest pre-condition

- Program seen as a predicate transformer
- Given a function $s$, a pre-condition $Pre$ and a post-condition $Post$
- We start from $Post$ at the end of the function and go backwards
- At each step, we have a property $Q$ and a statement $s$, and compute the \textit{weakest pre-condition} $P$ such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property $P$, we must prove $Pre \Rightarrow P$. 
Deductive Verification - Hoare logic

Some rules

- **Assignment**
  \[ WP(x=e, Q) = Q[x ← e] \]

- **Sequence**
  \[ WP(s_1; s_2, Q) = WP(s_1, WP(s_2, Q)) \]

- **Conditional**
  \[ WP(\text{if } (e) s_1 \text{ else } s_2, Q) = e \Rightarrow WP(s_1, Q) \land \neg e \Rightarrow WP(s_2, Q) \]

- **While**
  \[ WP(\text{while } (e) s, Q) = \]
  \[ l \land \forall \omega . l \Rightarrow (e \Rightarrow WP(s, l) \land \neg e \Rightarrow Q) \]
Deductive Verification - Pointers and Memory

Memory Model

Issue
How can we represent memory operations (*x, a[i]=42,...) in the logic

▶ If too low-level (a big array of bytes), proof obligations are intractable.
▶ If too abstract, some C constructions can not be represented (arbitrary pointer casts, aliasing)
▶ Standard solution (Burstal-Bornat): replace struct’s components by a function
**Issue**
The same memory location can be accessed through different means:

```c
int y;
int* yptr = &y;
*yptr = 3;
/*@ assert y == 3; */
```

- Again, supposing that any two pointers can be aliases would lead to intractable proof obligations.
- Memory is separated in disjoint regions
- Some hypotheses are done (as additional pre-conditions)
What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)
What is Jessie Useful for?

Usage
- Proof of functional properties of the program
- Modular verification (function per function)

Limitations
- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care
What is Jessie Useful for?

Usage
▶ Proof of functional properties of the program
▶ Modular verification (function per function)

Limitations
▶ Cast between pointers and integers
▶ Limited support for union type
▶ Aliasing requires some care
What is Jessie Useful for?

**Usage**

- Proof of functional properties of the program
- Modular verification (function per function)

**Limitations**

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care
Deductive Verification - Jessie Plugin

What is Jessie Useful for?

**Usage**
- Proof of functional properties of the program
- Modular verification (function per function)

**Limitations**
- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care
What is Jessie Useful for?

Usage
- Proof of functional properties of the program
- Modular verification (function per function)

Limitations
- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care
What is Jessie Useful for?

Usage
- Proof of functional properties of the program
- Modular verification (function per function)

Limitations
- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care
Deductive Verification - Jessie Plugin

From Frama-C to Theorem Provers

Why file

Jessie

Frama-C

Why

Verification conditions

Automated provers:
Alt-ergo
Simplify
Z3
...

Proof assistants:
Coq
Isabelle
PVS

C file
In practice

- Launch GUI:
  
  frama-c -jessie file.c

- Batch processing with alt-ergo:
  
  frama-c -jessie -jessie-atp alt-ergo file.c

- Generate Coq file (to be completed interactively):
  
  frama-c -jessie -jessie-atp coq file.c

- Concentrate on functional properties:
  
  frama-c -jessie -jessie-behavior default file.c

- Concentrate on safety properties:
  
  frama-c -jessie -jessie-behavior safety file.c