

Frama-C WP Tutorial

Virgile Prevosto, Nikolay Kosmatov and Julien Signoles

June 11th, 2013

```
long n;  
for (i = 0; i < n; i++)  
  C[i] = 0;  
tmp2 = ...  
... of the
```

```
tmp2[i] = 0; if (i < (n-1)) else if (tmp1[i]) >= 1) << (n-1) - i; else tmp2[i] = (tmp1[i] + 1) * Then the second part looks like the first one: tmp2[i] = 0; k = 5; k--> tmp1[i][k] += m2[i][k] * tmp2[k][j]; The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1  
l = 3; tmp1[i][l] >= 1; Final rounding: tmp2[i][l] is now represented on 9 bits. *if (tmp1[i][l] < -256) m2[i][l] = -256; else if (tmp1[i][l] > 255) m2[i][l] = 255; else m2[i][l] = tmp1[i][l];
```



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
- ▶ ...



long n
for 0 <=
C1) if m
tmp2 =
of the

tmp2[0] = 1 <= (n-1) else if tmp[0] >= 1 <= (n-1) tmp[0] = 1 <= (n-1) + tmp2[0] = tmp[0] + 1
tmp[0] = 0; k = 0; while tmp[0] != m2[0][k] tmp[0][k] = m2[0][k] tmp[0][k] = The [k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
l = 1; tmp[0][0] >= 1; Final rounding: tmp2[0] is now represented on 9 bits: *if tmp[0][0] < -256 m2[0][0] = -256 else if tmp[0][0] > 255 m2[0][0] = 255 else tmp[0][0] = tmp[0][0]

In this tutorial, we will see

- ▶ how to specify a C program with ACSL
- ▶ how to prove it automatically with Frama-C/WP
- ▶ how to understand and fix proof failures

long n;
for (i = 0; i < n; i++)
C[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; k = 0; while (tmp2[i] != 0) { tmp2[i] = tmp2[i] * 2; k++; }
tmp2[i] = 0; k = 0; while (tmp2[i] != 0) { tmp2[i] = tmp2[i] * 2; k++; }
The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP2) = MC2*(MC1*M1) = MC2*M1*MC1
l = 1; tmp2[i][l] += 1; } Final rounding: tmp2[i][l] is now represented on 9 bits: *if (tmp2[i][l] < -256) tmp2[i][l] = -256; else if (tmp2[i][l] > 255) tmp2[i][l] = 255; else tmp2[i][l] = tmp2[i][l];



Presentation of Frama-C

Context

First steps

Frama-C plugins

Basic function contract

A little bit of background

ACSL and WP

Specifying side-effects

Loops

Background

Loop invariants in ACSL

Loop termination

Advanced contracts

Behaviors

User-defined predicates

(long n) ...
for (k = 0; k < n; k++) ...
if (m[k] < 0) ...
tmp2 = ...
of the ...

tmp2[0] = 1 << (n-1) else if (tmp1[0]) >> 1 << (n-1) tmp2[0] = 1 << (n-1) - 1; else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: */
tmp1[0][0] = 0; k = k + 1; tmp1[0][k] += mc2[0][k] * tmp2[0][k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
l = 1; tmp1[0][l] >>= 1; /* Final rounding: tmp2[0][l] is now represented on 9 bits. *if (tmp1[0][l] < -256) m2[0][l] = -256; else if (tmp1[0][l] > 255) m2[0][l] = 255; else m2[0][l] = tm...



A brief history

- ▶ 90's: CAVEAT, an Hoare logic-based tool for C programs at CEA
- ▶ 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- ▶ 2002: Why and its C front-end Caduceus (at INRIA)
- ▶ 2006: Joint project to write a successor to CAVEAT and Caduceus
- ▶ 2008: First public release of Frama-C (Hydrogen)
- ▶ 2010: start of Device-Soft project between Fraunhofer FIRST (now FOKUS) and CEA LIST
- ▶ today:
 - ▶ Frama-C Fluorine (v9)
 - ▶ Multiple projects around the platform
 - ▶ A growing community of users
 - ▶ and of plug-ins developers



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>

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)`



On Linux

- ▶ On Debian, Ubuntu, Fedora, Gentoo, OpenSuse, Linux Mint, ...
- ▶ Compile from sources using OCaml package managers:
 - ▶ Godi
(<http://godi.camlcity.org/godi/index.html>)
 - ▶ Opam (<http://opam.ocamlpro.com/>)

On Windows

- ▶ Godi
- ▶ Wodi (<http://wodi.forge.ocamlcore.org/>)

On Mac OS X

- ▶ Binary package available
- ▶ Source compilation through homebrew.



Manuals

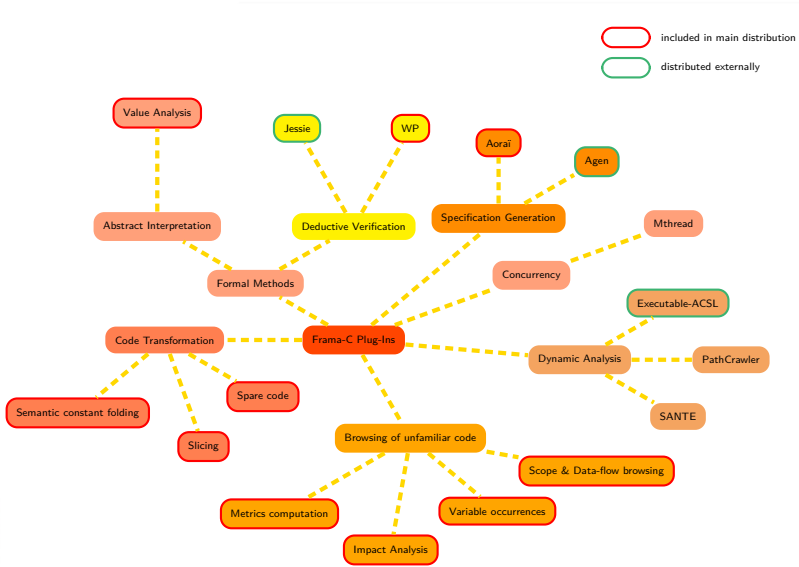
- ▶ `http://frama-c.com/support.html`
- ▶ In directory
`$(frama-c -print-share-path)/manuals`
- ▶ inline help (`frama-c -kernel-help`,
`frama-c -plugin-help`)

Support

- ▶ `frama-c-discuss@gforge.inria.fr`
- ▶ tag `frama-c` on `http://stackoverflow.com`



Main plug-ins



```

long n;
for (i = 0; i < n; i++)
  C[i] = 0;
tmp2[0] = 0;

```



External plugins

- ▶ Taster (coding rules, Atos/Airbus, Delmas &al., ERTS 2010)
- ▶ Dassault's internal plug-ins (Pariante & Ledinet, FoVeOOs 2010)
- ▶ Fan-C (flow dependencies, Atos/Airbus, Duprat &al., ERTS 2012)
- ▶ Simple Concurrency plug-in (Adelard, first release in 2013)
- ▶ Various academic experiments (mostly security and/or concurrency related)



long n
for n
ct; if m
tmp2
of d

tmp2[0] = (n < (n-1) ? n : tmp[1]); // if (n < (n-1) ? n : tmp[1])
 tmp[1] = 0; k = 5; k = tmp[1]; // m2[0][k] = tmp2[k]; // The [k] coefficient of the matrix product MC2*TMP2, that is *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1
 // = 1; // Final rounding: tmp2[0] is now represented on 3 bits: if (tmp[1] < 256) m2[0] = 256; else if (tmp[1] > 255) m2[0] = 255; else if (tmp[1] > 254) m2[0] = 254; else if (tmp[1] > 253) m2[0] = 253; else if (tmp[1] > 252) m2[0] = 252; else if (tmp[1] > 251) m2[0] = 251; else if (tmp[1] > 250) m2[0] = 250; else if (tmp[1] > 249) m2[0] = 249; else if (tmp[1] > 248) m2[0] = 248; else if (tmp[1] > 247) m2[0] = 247; else if (tmp[1] > 246) m2[0] = 246; else if (tmp[1] > 245) m2[0] = 245; else if (tmp[1] > 244) m2[0] = 244; else if (tmp[1] > 243) m2[0] = 243; else if (tmp[1] > 242) m2[0] = 242; else if (tmp[1] > 241) m2[0] = 241; else if (tmp[1] > 240) m2[0] = 240; else if (tmp[1] > 239) m2[0] = 239; else if (tmp[1] > 238) m2[0] = 238; else if (tmp[1] > 237) m2[0] = 237; else if (tmp[1] > 236) m2[0] = 236; else if (tmp[1] > 235) m2[0] = 235; else if (tmp[1] > 234) m2[0] = 234; else if (tmp[1] > 233) m2[0] = 233; else if (tmp[1] > 232) m2[0] = 232; else if (tmp[1] > 231) m2[0] = 231; else if (tmp[1] > 230) m2[0] = 230; else if (tmp[1] > 229) m2[0] = 229; else if (tmp[1] > 228) m2[0] = 228; else if (tmp[1] > 227) m2[0] = 227; else if (tmp[1] > 226) m2[0] = 226; else if (tmp[1] > 225) m2[0] = 225; else if (tmp[1] > 224) m2[0] = 224; else if (tmp[1] > 223) m2[0] = 223; else if (tmp[1] > 222) m2[0] = 222; else if (tmp[1] > 221) m2[0] = 221; else if (tmp[1] > 220) m2[0] = 220; else if (tmp[1] > 219) m2[0] = 219; else if (tmp[1] > 218) m2[0] = 218; else if (tmp[1] > 217) m2[0] = 217; else if (tmp[1] > 216) m2[0] = 216; else if (tmp[1] > 215) m2[0] = 215; else if (tmp[1] > 214) m2[0] = 214; else if (tmp[1] > 213) m2[0] = 213; else if (tmp[1] > 212) m2[0] = 212; else if (tmp[1] > 211) m2[0] = 211; else if (tmp[1] > 210) m2[0] = 210; else if (tmp[1] > 209) m2[0] = 209; else if (tmp[1] > 208) m2[0] = 208; else if (tmp[1] > 207) m2[0] = 207; else if (tmp[1] > 206) m2[0] = 206; else if (tmp[1] > 205) m2[0] = 205; else if (tmp[1] > 204) m2[0] = 204; else if (tmp[1] > 203) m2[0] = 203; else if (tmp[1] > 202) m2[0] = 202; else if (tmp[1] > 201) m2[0] = 201; else if (tmp[1] > 200) m2[0] = 200; else if (tmp[1] > 199) m2[0] = 199; else if (tmp[1] > 198) m2[0] = 198; else if (tmp[1] > 197) m2[0] = 197; else if (tmp[1] > 196) m2[0] = 196; else if (tmp[1] > 195) m2[0] = 195; else if (tmp[1] > 194) m2[0] = 194; else if (tmp[1] > 193) m2[0] = 193; else if (tmp[1] > 192) m2[0] = 192; else if (tmp[1] > 191) m2[0] = 191; else if (tmp[1] > 190) m2[0] = 190; else if (tmp[1] > 189) m2[0] = 189; else if (tmp[1] > 188) m2[0] = 188; else if (tmp[1] > 187) m2[0] = 187; else if (tmp[1] > 186) m2[0] = 186; else if (tmp[1] > 185) m2[0] = 185; else if (tmp[1] > 184) m2[0] = 184; else if (tmp[1] > 183) m2[0] = 183; else if (tmp[1] > 182) m2[0] = 182; else if (tmp[1] > 181) m2[0] = 181; else if (tmp[1] > 180) m2[0] = 180; else if (tmp[1] > 179) m2[0] = 179; else if (tmp[1] > 178) m2[0] = 178; else if (tmp[1] > 177) m2[0] = 177; else if (tmp[1] > 176) m2[0] = 176; else if (tmp[1] > 175) m2[0] = 175; else if (tmp[1] > 174) m2[0] = 174; else if (tmp[1] > 173) m2[0] = 173; else if (tmp[1] > 172) m2[0] = 172; else if (tmp[1] > 171) m2[0] = 171; else if (tmp[1] > 170) m2[0] = 170; else if (tmp[1] > 169) m2[0] = 169; else if (tmp[1] > 168) m2[0] = 168; else if (tmp[1] > 167) m2[0] = 167; else if (tmp[1] > 166) m2[0] = 166; else if (tmp[1] > 165) m2[0] = 165; else if (tmp[1] > 164) m2[0] = 164; else if (tmp[1] > 163) m2[0] = 163; else if (tmp[1] > 162) m2[0] = 162; else if (tmp[1] > 161) m2[0] = 161; else if (tmp[1] > 160) m2[0] = 160; else if (tmp[1] > 159) m2[0] = 159; else if (tmp[1] > 158) m2[0] = 158; else if (tmp[1] > 157) m2[0] = 157; else if (tmp[1] > 156) m2[0] = 156; else if (tmp[1] > 155) m2[0] = 155; else if (tmp[1] > 154) m2[0] = 154; else if (tmp[1] > 153) m2[0] = 153; else if (tmp[1] > 152) m2[0] = 152; else if (tmp[1] > 151) m2[0] = 151; else if (tmp[1] > 150) m2[0] = 150; else if (tmp[1] > 149) m2[0] = 149; else if (tmp[1] > 148) m2[0] = 148; else if (tmp[1] > 147) m2[0] = 147; else if (tmp[1] > 146) m2[0] = 146; else if (tmp[1] > 145) m2[0] = 145; else if (tmp[1] > 144) m2[0] = 144; else if (tmp[1] > 143) m2[0] = 143; else if (tmp[1] > 142) m2[0] = 142; else if (tmp[1] > 141) m2[0] = 141; else if (tmp[1] > 140) m2[0] = 140; else if (tmp[1] > 139) m2[0] = 139; else if (tmp[1] > 138) m2[0] = 138; else if (tmp[1] > 137) m2[0] = 137; else if (tmp[1] > 136) m2[0] = 136; else if (tmp[1] > 135) m2[0] = 135; else if (tmp[1] > 134) m2[0] = 134; else if (tmp[1] > 133) m2[0] = 133; else if (tmp[1] > 132) m2[0] = 132; else if (tmp[1] > 131) m2[0] = 131; else if (tmp[1] > 130) m2[0] = 130; else if (tmp[1] > 129) m2[0] = 129; else if (tmp[1] > 128) m2[0] = 128; else if (tmp[1] > 127) m2[0] = 127; else if (tmp[1] > 126) m2[0] = 126; else if (tmp[1] > 125) m2[0] = 125; else if (tmp[1] > 124) m2[0] = 124; else if (tmp[1] > 123) m2[0] = 123; else if (tmp[1] > 122) m2[0] = 122; else if (tmp[1] > 121) m2[0] = 121; else if (tmp[1] > 120) m2[0] = 120; else if (tmp[1] > 119) m2[0] = 119; else if (tmp[1] > 118) m2[0] = 118; else if (tmp[1] > 117) m2[0] = 117; else if (tmp[1] > 116) m2[0] = 116; else if (tmp[1] > 115) m2[0] = 115; else if (tmp[1] > 114) m2[0] = 114; else if (tmp[1] > 113) m2[0] = 113; else if (tmp[1] > 112) m2[0] = 112; else if (tmp[1] > 111) m2[0] = 111; else if (tmp[1] > 110) m2[0] = 110; else if (tmp[1] > 109) m2[0] = 109; else if (tmp[1] > 108) m2[0] = 108; else if (tmp[1] > 107) m2[0] = 107; else if (tmp[1] > 106) m2[0] = 106; else if (tmp[1] > 105) m2[0] = 105; else if (tmp[1] > 104) m2[0] = 104; else if (tmp[1] > 103) m2[0] = 103; else if (tmp[1] > 102) m2[0] = 102; else if (tmp[1] > 101) m2[0] = 101; else if (tmp[1] > 100) m2[0] = 100; else if (tmp[1] > 99) m2[0] = 99; else if (tmp[1] > 98) m2[0] = 98; else if (tmp[1] > 97) m2[0] = 97; else if (tmp[1] > 96) m2[0] = 96; else if (tmp[1] > 95) m2[0] = 95; else if (tmp[1] > 94) m2[0] = 94; else if (tmp[1] > 93) m2[0] = 93; else if (tmp[1] > 92) m2[0] = 92; else if (tmp[1] > 91) m2[0] = 91; else if (tmp[1] > 90) m2[0] = 90; else if (tmp[1] > 89) m2[0] = 89; else if (tmp[1] > 88) m2[0] = 88; else if (tmp[1] > 87) m2[0] = 87; else if (tmp[1] > 86) m2[0] = 86; else if (tmp[1] > 85) m2[0] = 85; else if (tmp[1] > 84) m2[0] = 84; else if (tmp[1] > 83) m2[0] = 83; else if (tmp[1] > 82) m2[0] = 82; else if (tmp[1] > 81) m2[0] = 81; else if (tmp[1] > 80) m2[0] = 80; else if (tmp[1] > 79) m2[0] = 79; else if (tmp[1] > 78) m2[0] = 78; else if (tmp[1] > 77) m2[0] = 77; else if (tmp[1] > 76) m2[0] = 76; else if (tmp[1] > 75) m2[0] = 75; else if (tmp[1] > 74) m2[0] = 74; else if (tmp[1] > 73) m2[0] = 73; else if (tmp[1] > 72) m2[0] = 72; else if (tmp[1] > 71) m2[0] = 71; else if (tmp[1] > 70) m2[0] = 70; else if (tmp[1] > 69) m2[0] = 69; else if (tmp[1] > 68) m2[0] = 68; else if (tmp[1] > 67) m2[0] = 67; else if (tmp[1] > 66) m2[0] = 66; else if (tmp[1] > 65) m2[0] = 65; else if (tmp[1] > 64) m2[0] = 64; else if (tmp[1] > 63) m2[0] = 63; else if (tmp[1] > 62) m2[0] = 62; else if (tmp[1] > 61) m2[0] = 61; else if (tmp[1] > 60) m2[0] = 60; else if (tmp[1] > 59) m2[0] = 59; else if (tmp[1] > 58) m2[0] = 58; else if (tmp[1] > 57) m2[0] = 57; else if (tmp[1] > 56) m2[0] = 56; else if (tmp[1] > 55) m2[0] = 55; else if (tmp[1] > 54) m2[0] = 54; else if (tmp[1] > 53) m2[0] = 53; else if (tmp[1] > 52) m2[0] = 52; else if (tmp[1] > 51) m2[0] = 51; else if (tmp[1] > 50) m2[0] = 50; else if (tmp[1] > 49) m2[0] = 49; else if (tmp[1] > 48) m2[0] = 48; else if (tmp[1] > 47) m2[0] = 47; else if (tmp[1] > 46) m2[0] = 46; else if (tmp[1] > 45) m2[0] = 45; else if (tmp[1] > 44) m2[0] = 44; else if (tmp[1] > 43) m2[0] = 43; else if (tmp[1] > 42) m2[0] = 42; else if (tmp[1] > 41) m2[0] = 41; else if (tmp[1] > 40) m2[0] = 40; else if (tmp[1] > 39) m2[0] = 39; else if (tmp[1] > 38) m2[0] = 38; else if (tmp[1] > 37) m2[0] = 37; else if (tmp[1] > 36) m2[0] = 36; else if (tmp[1] > 35) m2[0] = 35; else if (tmp[1] > 34) m2[0] = 34; else if (tmp[1] > 33) m2[0] = 33; else if (tmp[1] > 32) m2[0] = 32; else if (tmp[1] > 31) m2[0] = 31; else if (tmp[1] > 30) m2[0] = 30; else if (tmp[1] > 29) m2[0] = 29; else if (tmp[1] > 28) m2[0] = 28; else if (tmp[1] > 27) m2[0] = 27; else if (tmp[1] > 26) m2[0] = 26; else if (tmp[1] > 25) m2[0] = 25; else if (tmp[1] > 24) m2[0] = 24; else if (tmp[1] > 23) m2[0] = 23; else if (tmp[1] > 22) m2[0] = 22; else if (tmp[1] > 21) m2[0] = 21; else if (tmp[1] > 20) m2[0] = 20; else if (tmp[1] > 19) m2[0] = 19; else if (tmp[1] > 18) m2[0] = 18; else if (tmp[1] > 17) m2[0] = 17; else if (tmp[1] > 16) m2[0] = 16; else if (tmp[1] > 15) m2[0] = 15; else if (tmp[1] > 14) m2[0] = 14; else if (tmp[1] > 13) m2[0] = 13; else if (tmp[1] > 12) m2[0] = 12; else if (tmp[1] > 11) m2[0] = 11; else if (tmp[1] > 10) m2[0] = 10; else if (tmp[1] > 9) m2[0] = 9; else if (tmp[1] > 8) m2[0] = 8; else if (tmp[1] > 7) m2[0] = 7; else if (tmp[1] > 6) m2[0] = 6; else if (tmp[1] > 5) m2[0] = 5; else if (tmp[1] > 4) m2[0] = 4; else if (tmp[1] > 3) m2[0] = 3; else if (tmp[1] > 2) m2[0] = 2; else if (tmp[1] > 1) m2[0] = 1; else if (tmp[1] > 0) m2[0] = 0; else m2[0] = tmp[1];

Presentation of Frama-C

Context

First steps

Frama-C plugins

Basic function contract

A little bit of background

ACSL and WP

Specifying side-effects

Loops

Background

Loop invariants in ACSL

Loop termination

Advanced contracts

Behaviors

User-defined predicates

long n;
for (i = 0; i < n; i++)
 tmp2[i] = i;

tmp2[0] = 0; // (n-1) else tmp2[0] = 1; // (n-1) - 1; else tmp2[0] = tmp2[0]; // Then the second part looks like the first one: m[i] = m[i] * k; // k = 0; k = 1; tmp2[0] = m[0][0] * tmp2[0]; // The [i] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp2[0] >= 1; // Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp2[0] < -256) m2[0] = -256; else if (tmp2[0] > 255) m2[0] = 255; else m2[0] = tmp2[0];



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)



```
/*@ requires R;
   ensures E; */
int f(int* x) {
```

```
S_1;

S_2;

}
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \Rightarrow \{P\}S\{Q\}$$

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$




```
/*@ requires R;
   ensures E; */
int f(int* x) {
```

```
S_1;
```

```
S_2;
```

```
/*@assert E; */
}
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \Rightarrow \{P\}S\{Q\}$$

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$



```
/*@ requires R;
   ensures E; */
int f(int* x) {
```

```
S_1;
```

```
/*@assert wp(S_2, E); */
S_2;
```

```
/*@assert E; */
}
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\}$$

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$



```
/*@ requires R;
   ensures E; */
int f(int* x) {
```

```
/*@assert
   wp(S_1, wp(S_2, E)); */
S_1;
```

```
/*@assert wp(S_2, E); */
S_2;
```

```
/*@assert E; */
}
```

▶ Hoare Triples:

$$\{P\}S\{Q\}$$

▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\}$$

▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$



A first example

```

#include "limits.h"
// returns the maximum of x and y
int max ( int x, int y ) {
    if ( x >=y )
        return x ;
    return y ;
}
  
```



Credits

- ▶ Loïc Correnson
- ▶ Zaynah Dargaye
- ▶ Anne Pacalet
- ▶ François Bobot
- ▶ a few others

Basic usage

- ▶ `frama-c-gui -wp file.c`
- ▶ WP tab on the GUI
- ▶ Inspect (failed) proof obligation
- ▶ <http://frama-c.com/download/wp-manual.pdf>



Avoiding run-time errors

Example

```
// returns the absolute value of x
int abs ( int x ) {
    if ( x >=0 )
        return x ;
    return -x ;
}
```

Command

- ▶ `frama-c-gui -pp-annot -wp -wp-rte abs.c`
- ▶ or use switch directly in GUI



Dealing with pointers

Example

```

// returns the maximum of *p and *q
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
  
```

Main ingredients

- ▶ built-in predicate `\valid(...)`
- ▶ **assigns** clause



Example

```
// swap the content of both arguments
void swap(int* p, int* q) {
    int tmp = *q;
    *q = *p;
    *p = tmp;
}
```




```
/*@ requires R_1;
   ensures E_1;
   assigns A;
```

```
*/
void g();
```

```
/*@ requires R_2;
   ensures E_2;
```

```
*/
void f() {
  S_1;
  g();
  S_2;
}
```

- ▶ Contract as a cut

- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall \text{State}, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in \text{State} \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;
   ensures E_1;
   assigns A;
```

```
*/
void g ();
```

```
/*@ requires R_2;
   ensures E_2;
```

```
*/
void f () {
    S_1;
    g ();
    S_2;
}
```

- ▶ Contract as a cut
- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall \text{State}, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in \text{State} \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;
   ensures E_1;
   assigns A;
```

```
*/
void g();
```

```
/*@ requires R_2;
   ensures E_2;
```

```
*/
void f() {
    S_1;
    g();
    S_2;
}
```

- ▶ Contract as a cut
- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall State, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$\forall x \in State \setminus A, g$ does not change x



Function Calls

```
/*@ requires R_1;  
   ensures E_1;  
   assigns A;
```

```
*/  
void g ();
```

```
/*@ requires R_2;  
   ensures E_2;
```

```
*/  
void f () {  
    S_1;  
    g ();  
    S_2;  
}
```

- ▶ Contract as a cut
- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall State, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$\forall x \in State \setminus A, g$ does not change x



```
/*@ requires R_1;
   ensures E_1;
   assigns A;
```

```
*/
void g ();
```

```
/*@ requires R_2;
   ensures E_2;
```

```
*/
void f () {
  S_1;
  g ();
  S_2;
}
```

- ▶ Contract as a cut
- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall State, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in State \setminus A, g \text{ does not change } x$$



Function call: example

```

#include "limits.h"
/*@
  requires \valid(p) && \valid(q);
  ensures \result >= *p && \result >= *q;
  ensures \result == *p || \result == *q;
  assigns \nothing;
*/
int max ( int* p, int* q );

```

(long int
for 0 <=
C1) if (m
tmp2 =
of the

tmp2[0] = 1 <= (n1 - 1) else if (tmp1[0]) >= 1 <= (n1 - 1) tmp2[0] = 1 <= (n1 - 1) + tmp1[0] = tmp1[0]; 7. Then the second part takes for the first part
tmp1[0] = 0; k = 8; k = 9; tmp1[0] = mc2[0][k] * tmp2[0][k]; 8. The [i][j] coefficient of the matrix product MC2 * TMP2, that is, *MC2*(TMP1) = MC2*(M1) = MC2*(M1) * MC1
1 = 1; tmp1[0] >= 1; 9. Final rounding: tmp2[0] is now represented on 9 bits: *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];



Function call: example (cont'd)

```

/*@
  requires \valid(p) && \valid(q);
  assigns *x, *y;
  ensures *x == \at(*y,Pre);
  ensures *y == \at(*x,Pre);
*/
void swap(int* x, int* y);

// ensures that *high contains
// the maximum of the two values.
int max_swap( int* low, int* high ) {
  if (*high != max(low,high)) swap(low,high);
}

```




```
/*@ requires R;
   ensures E;
*/
void f () {
  S_1;
```

```
while (e) { B }
S_2;
}
```

- ▶ Need to capture effects of **all** loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step) PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



```

/*@ requires R;
   ensures E;
*/
void f () {
  S_1;

  /*@loop invariant I;

*/
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



```

/*@ requires R;
   ensures E;
*/
void f () {
  S_1;

  /*@loop invariant I;

  */
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



```

/*@ requires R;
   ensures E;
*/
void f() {
  S_1;

  /*@loop invariant I;

  */
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



```

/*@ requires R;
   ensures E;
*/
void f () {
  S_1;

  /*@loop invariant I;

  */
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 - $State \setminus A, B$ does not change x



```

/*@ requires R;
   ensures E;
*/
void f () {
  S_1;

  /*@loop invariant I;
   loop assigns A;
  */
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



Loops: example

```

// returns a non-zero value iff all elements
// in a given array t of n integers are zeros
int all_zeros(int t[], int n) {
    int k;
    for(k = 0; k < n; k++)
        if (t[k] != 0)
            return 0;
    return 1;
}

```



Loop invariants - some hints

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

- ▶ identify **locations modified in the loop**
 - ▶ 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.



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 right 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



Presentation of Frama-C

Context

First steps

Frama-C plugins

Basic function contract

A little bit of background

ACSL and WP

Specifying side-effects

Loops

Background

Loop invariants in ACSL

Loop termination

Advanced contracts

Behaviors

User-defined predicates

long na
for (i
C1) if (m
tmp2 =
of the

tmp2[0] = 1 << (nbl - 1) else if (tmp1[0] >= 1) << (nbl - 1) tmp2[0] = 1 << (nbl - 1) + tmp2[0] + tmp1[0]; /* Then the second part looks like the first one: */
tmp1[0][0] = 0; k = 0; k++ tmp1[0][k] += mc2[0][k] * tmp2[0][k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
l = 1; tmp1[0][l] >= 1; /* Final rounding: tmp2[0][l] is now represented on 9 bits. *if (tmp1[0][l] < -256) m2[0][l] = -256; else if (tmp1[0][l] > 255) m2[0][l] = 255; else m2[0][l] = tm



Specification by cases

- ▶ Global precondition (**requires**) and postcondition (**ensures**, **assigns**) applies to all cases
- ▶ Behaviors refine global contract in particular cases
- ▶ For each case (each **behavior**)
 - ▶ the subdomain is defined by **assumes** clause
 - ▶ can give additional constraints with local **requires** clauses
 - ▶ 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



Using behaviors: example

```

/* input: a sorted array a, its length,
   and a value key to search.
   output: index of a cell which contains key,
   or -1 if key is not present in the array.
*/
int binary_search(int* a, int length, int key) {
  int low = 0, high = length - 1;
  while (low<=high) {
    int mid = (low+high)/2;
    if (a[mid] == key) return mid;
    if (a[mid] < key) { low = mid+1; }
    else { high = mid - 1; }
  }
  return -1;
}

```



A look a C strings

From C std library

```
#include "limits.h"
```

```
typedef unsigned int size_t;
```

```
void* memcpy(void* dest, void* src, size_t length);
```

```
size_t strlen(char* s);
```

```
char* strcpy(char *s1, const char* s2);
```

```
char *strncpy(char *s1, const char *s2, size_t n
```

