Copilot : Traceability and Verification of a Low Level Automatically Generated C Source Code

Georges-Axel Jaloyan

École Normale Supérieure, NASA Langley Research center, National Institute of Aerospace

August 12, 2015

1. Preliminaries
   - Copilot language
   - ACSL
   - Copilot toolchain
2. Working on the backend
3. Applications

Copilot language

Copilot is an *EDSL* (embedded domain specific language), embedded in *Haskell* and used for writing *runtime monitors* for hard real-time, distributed, reactive systems written in C.

A Copilot program, can either be :

- compiled to C using two back-ends : SBV, ATOM
- interpreted
- analyzed using static analysis tools (CBMC, Kind)

Copilot syntax

A program is a list of streams that can be either external or internal which are defined by mutually recursive stream equations.

Each stream has a type which can be `Bool`, `Int8`, `Int16`, `Int32`, `Int64`, `Word8`, `Word16`, `Word32`, `Word64`, `Float`, `Double`.

```
x :: Stream Word16
x = 0
-- x = {0, 0, 0, ...}
y :: Stream Bool
y = x `mod` 2 == 0
-- y = {T, T, ...}
nats :: Stream Word64
nats = [0] ++ (1 + nats)
-- nats = {0,1,2, ..., 2^64-1, 0, 1, ..}
```

Operators

Each operator and constant has been lifted to Streams (working pointwise).

Two temporal operations working on Streams :

- ++ : which prepends a finite list to a Stream

```
(++) :: [a] -> Stream a -> Stream a
```

- drop : which drops a finite number of elements at the beginning of a Stream

```
drop :: Int -> Stream a -> Stream a
```

Casts and unsafe casts are also provided :

```
cast :: (Typed a, Typed b) => Stream a -> Stream b
unsafeCast :: (Typed a, Typed b) => Stream a -> Stream b
```

Examples

Fibonacci sequence :

```
fib :: Stream Word64
fib = [1,1] ++ (fib + drop 1 fib)
-- fib = {1,1,2,3,5,8,13,...,
--         12200160415121876738,
--    /!\ 1293530146158671551,...}
```

Interaction

Sensors :

- Sample external variables.

```
extern :: Typed a => String -> Maybe [a] -> Stream a
```

Example :

```
unsigned long long int x;
```

```
x :: Stream Word64
x = extern "x" (Just [0,0..])

x2 = externW64 "x" Nothing
```

Interaction

Sensors :

- Sample external variables.
- Sample external arrays.

```
externArray :: (Typed a, Typed b, Integral a) =>
String -> Stream a -> Int -> Maybe [[a]] -> Stream b
```

Example :

```
unsigned long long int tab[1000];
```

```
-- nat = [0] ++ (nats + 1)
x :: Stream Word64
x = externArray "tab" nats 1000 Nothing

x2 = externArrayW64 "tab" nats 1000 Nothing
```

Interaction

Sensors :

- Sample external variables.
- Sample external arrays.
- Sample external functions.

```
externFun :: Typed a =>
String -> [FunArg] -> Maybe [a] -> Stream a
```

Example :

```
double sin(double a); //from math.h

x :: Stream Double
x = externDouble "x" Nothing

sinx = externFun "sin" [arg x] Nothing
```

Interaction

Sensors :

- Sample external variables.
- Sample external arrays.
- Sample external functions.

Interaction

Actuators :

- Triggers :

```
trigger ::
  String -> Stream Bool -> [TriggerArg] -> Spec
```

- Observers :

```
observer :: Typed a => String -> Stream a -> Spec
```

1. Preliminaries
   - Copilot language
   - ACSL
   - Copilot toolchain
2. Working on the backend
3. Applications

ACSL syntax

ACSL is a specification language for C programs. Those contracts are written according to the following example :

```
/*@ requires true
assigns \nothing
ensures \result >= x && \result >= y;
ensures \result == x || \result == y;
*/
int max (int x, int y) { return (x > y) ? x : y; }
```

Floyd-Hoare logic

A Floyd-Hoare triple is :
$\{P\}$ *prog* $\{Q\}$

- *prog* is a program fragment
- P and Q are logical assertions over program variables
- P is the precondition
- Q the postcondition

$\{P\}$ *prog* $\{Q\}$ holds iff

- P holds before the execution of *prog*
- Q holds after the execution of *prog*[1]

[1] Unless *prog* does not terminate or encounters an error.

Floyd-Hoare logic

Here is an example of a proof tree of a program[2]:

$$
\dfrac{\dfrac{\dfrac{}{\{true\}\ I \leftarrow I-1\ \ \{true\}}}{\{I \neq 0\}\ I \leftarrow I-1\ \ \{true\}}}{\dfrac{\{true\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ I \leftarrow I-1\ \ \{true \wedge \neg(I \neq 0)\}}{\{true\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ I \leftarrow I-1\ \ \{I = 0\}}}
$$

[2] A. Miné, *Semantics and application to program verification : Axiomatic semantics*, 2015.

Floyd-Hoare logic

The Floyd-Hoare logic does not take into account program termination:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathit{true}\}\ I \leftarrow I\ \ \{\mathit{true}\}}}{\{I \neq 0\}\ I \leftarrow I\ \ \{\mathit{true}\}}}{\{\mathit{true}\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ I \leftarrow I\ \ \{\mathit{true} \wedge \neg(I \neq 0)\}}}{\{\mathit{true}\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ I \leftarrow I\ \{I = 0\}}$$

Floyd-Hoare logic

Or even safety against runtime errors (we speak about partial correctness):

$$\dfrac{\dfrac{\dfrac{}{\{true\}\ \textbf{fail}\ \ \{true\}}}{\{I \neq 0\}\ \textbf{fail}\ \ \{true\}}}{\dfrac{\{true\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ \textbf{fail}\ \ \{true \wedge \neg(I \neq 0)\}}{\{true\}\ \texttt{while}\ I \neq 0\ \texttt{do}\ \textbf{fail}\ \{I = 0\}}}$$

More generally, any property is true after fail :

$$\dfrac{}{\{P\}\ \textbf{fail}\ \{Q\}}$$

Floyd-Hoare logic

It is nevertheless possible to prove total correctness by the following proof tree (ranking functions have to be provided):

$$\frac{\{P\}\ prog\ \{Q\} \qquad [P]\ prog\ [true]}{[P]\ prog\ [Q]}$$

Dijkstra's Weakest Liberal Precondition

We define the weakest liberal precondition : $wlp(prog, Q)$ which is defined as the most general condition such that $\{wlp(prog, Q)\}\ prog\ \{Q\}$ holds.

We can automate the computation of the precondition by induction on the syntax.

- $wlp(skip, P) = P$
- $wlp(fail, P) = true$
- $wlp(s; t, P) = wlp(s, wlp(t, P))$
- $wlp(X \leftarrow e, P) = P[e/X]$
- $wlp(\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t, P) = (e \Rightarrow wlp(s, P)) \wedge (\neg e \Rightarrow wlp(t, P))$

1. Preliminaries
   - Copilot language
   - ACSL
   - Copilot toolchain
2. Working on the backend
3. Applications

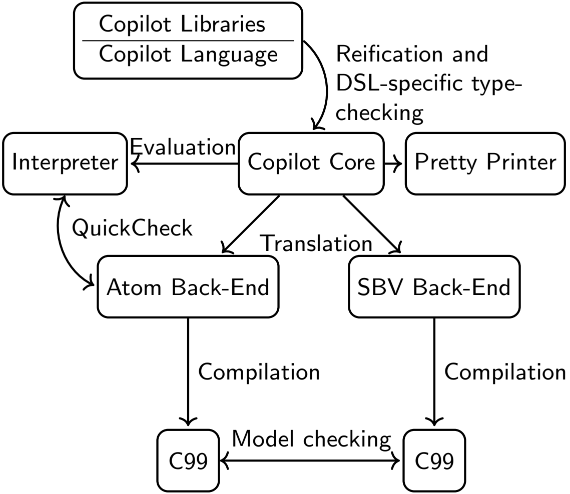


Figure: The Copilot toolchain[3]

[3] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, *Experience report: A do-it-yourself high-assurance compiler*, 2012.

1. Preliminaries

2. Working on the backend
   - Hand-written ACSL
   - ACSL generation
   - Bug finding

3. Applications

Hand written ACSL

```
import Copilot.Language.Reify
import Copilot.Language
import qualified Copilot.Compile.SBV as S

logic :: Stream Bool
logic = [True, False] ++ logic && drop 1 logic

spec :: Spec
spec = do
observer "obs1" logic

main = do
interpret 10 spec
reify spec >>= S.compile S.defaultParams --SBV Backend
```

```
/*@
requires ptr_2 < 0x0078;
requires \valid(queue_2 + (0..0x02U-1));
assigns \nothing;
ensures \result == ( queue_2[ptr_2 % 0x02U]
                && queue_2[(ptr_2 + 0x01U) % 0x02U]);
*/
SBool update_state_2(const SBool *queue_2
                    , const SWord16 ptr_2)
{
  const SWord16 s2 = ptr_2;
  const SWord16 s4 = (0x02U == 0)?s2:(s2%0x02U);
  const SBool   s5 = queue_2[s4];
  const SWord16 s7 = s2 + 0x0001U;
  const SWord16 s8 = (0x02U == 0)?s7:(s7%0x02U);
  const SBool   s9 = queue_2[s8];
  const SBool   s10 = s5 && s9;

  return s10;
}
```

```
frama-c -wp -wp-out .  -wp-prover PROVER

[wp] Proved goals:    19 / 19
Qed:              18   (4ms-4ms)
cvc4:              1   (150ms-150ms)

[wp] Proved goals:    19 / 19
Qed:              18   (4ms-4ms)
cvc3:              1   (90ms-90ms)

[wp] Proved goals:    19 / 19
Qed:              18   (4ms-8ms)
Alt-Ergo:          1   (3.5s-3.5s)  (248)

[wp] Proved goals:    19 / 19
Qed:              18   (4ms-4ms)
z3:                1   (20ms-20ms)
```

Bitwise version :

```
/*@
requires ptr_2 < 0x0078;
requires \valid(queue_2 + (0..0x02U-1));
assigns \nothing;
ensures \result == ( queue_2[ptr_2 % 0x02U]
& queue_2[(ptr_2 + 0x01U) % 0x02U]);
*/
SBool update_state_2(const SBool *queue_2
, const SWord16 ptr_2)
{
const SWord16 s2 = ptr_2;
const SWord16 s4 = (0x02U == 0)?s2:(s2%0x02U);
const SBool   s5 = queue_2[s4];
const SWord16 s7 = s2 + 0x0001U;
const SWord16 s8 = (0x02U == 0)?s7:(s7%0x02U);
const SBool   s9 = queue_2[s8];
const SBool   s10 = s5 & s9;

return s10;
}
```

```
frama-c -wp -wp-out .  -wp-prover PROVER

[wp] Proved goals:   15 / 16
Qed:              15  (4ms-4ms)
cvc4:              0  (interrupted: 1)

[wp] Proved goals:   15 / 16
Qed:              15  (4ms-4ms)
cvc3:              0  (unknown: 1)

[wp] Proved goals:   15 / 16
Qed:              15  (4ms-4ms)
Alt-Ergo:          0  (interrupted: 1)

[wp] Proved goals:   15 / 16
Qed:              15  (4ms-4ms)
z3:                0  (interrupted: 1)
----> Timeout after 30 seconds
```

Unsafe version :

```
/*@
requires \valid(queue_2 + (0..0x02U-1));
assigns \nothing;
ensures \result == ( queue_2[ptr_2 % 0x02U]
&& queue_2[(ptr_2 + 0x01U) % 0x02U]);
*/
SBool update_state_2(const SBool *queue_2
, const SWord16 ptr_2)
{
const SWord16 s2 = ptr_2;
const SWord16 s4 = (0x02U == 0)?s2:(s2%0x02U);
const SBool   s5 = queue_2[s4];
const SWord16 s7 = s2 + 0x0001U;
const SWord16 s8 = (0x02U == 0)?s7:(s7%0x02U);
const SBool   s9 = queue_2[s8];
const SBool   s10 = s5 && s9;

return s10;
}
```

```
frama-c -wp -wp-out .  -wp-prover PROVER

[wp] Proved goals:   18 / 19
Qed:             18  (4ms-4ms)
cvc4:             0  (interrupted: 1)

[wp] Proved goals:   18 / 19
Qed:             18  (4ms-4ms)
Alt-Ergo:         0  (interrupted: 1)

[wp] Proved goals:   18 / 19
Qed:             18  (4ms-4ms)
z3:               0  (unknown: 1)
----> NO TIMEOUT : unsafe
```

1 Preliminaries

2 Working on the backend
- Hand-written ACSL
- ACSL generation
- Bug finding

3 Applications

ACSL generation

The easiest way to do it is by induction on the syntax, when compiling the expression. Here is how the function ppACSL is constructed :

- `Const type value` $\rightarrow$ *show* `value`
- `Drop type i id` $\rightarrow$ *queue*_`id`[*ptr*_`id` + `i` *mod* (*length* `id`)]
- `ExternVar t name b` $\rightarrow$ *ext*_`name`
- `Var type name` $\rightarrow$ `name`
- `Op2 op e1 e2` $\rightarrow$ (*ppACSL* `e1`) *show* `op` (*ppACSL* `e2`)
- `Label t s e` $\rightarrow$ *ppACSL* `e`

ACSL generation

Nevertheless, some hacks :

- Let bindings have been deprecated.
- Abs are converted to $\backslash a \rightarrow sign\ a \times a$
- Sign to $\backslash x \rightarrow ((x > 0)\ ?\ 1 : ((x < 0)? -1 : 0))$
- Mux where branches have type Bool to
  $Mux\ e1\ e2\ e3 = (e2 \wedge e1) \vee (e3 \wedge \neg e1)$
- No bitwise operator are supported.

ACSL generation

Still some problems with frama-c :

- No global invariant : we have to split the dereferencing of the pointer into a black box that only do this.
- No math functions (such as sin, cos, exp, log, ...) : we have to do the same.

WP vs VA

How effective value analysis is ?

- Global invariants supported
- No lemma supported
- Safe ... for only one iteration of the main loop
- Does not really go well with external variables
- Requires access to all C source files of the project to say anything about one contract.

(Very bad) solution : unroll the infinite loop !

```
frama-c -val -main testing -slevel 10000000 *.h *.c
```

(Better) solution : forget about value analysis for the monitor.

Other changes

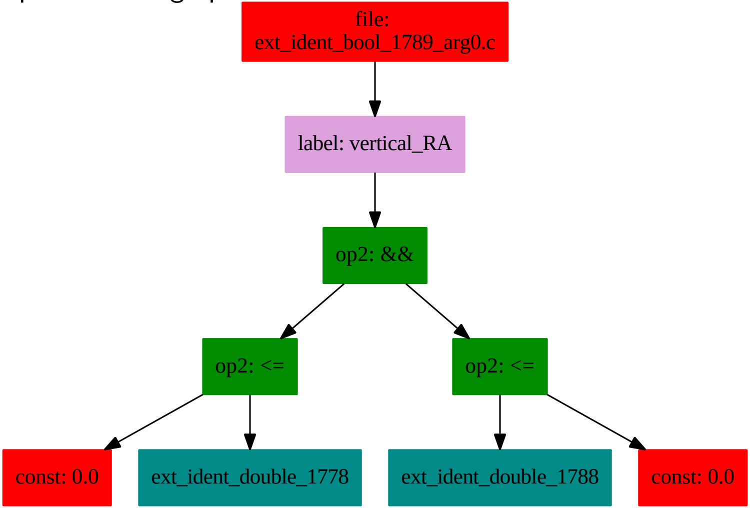
- Added m4 preprocessing
- Added CompCert for compiling the C source file generated
- Deprecated ATOM
- Added a dot graph generation for each source file.

Dot File: "ext_ident_bool_1789_arg0.c"

```
/*@
assigns \nothing;
ensures \result ==
      ((((((0.0) <= (ext_ident_double_1778)))
      && (((ext_ident_double_1788) <= (0.0))))));
*/
SBool ext_ident_bool_1789_arg0
(const SDouble ext_ident_double_1778,
const SDouble ext_ident_double_1788)
{
  const SDouble s0 = ext_ident_double_1778;
  const SDouble s14 = ext_ident_double_1788;
  const SBool   s25 = 0.0 <= s0;
  const SBool   s26 = s14 <= 0.0;
  const SBool   s27 = s25 && s26;
  const SBool   s28 = s27 /* vertical_RA */;
  return s28;
}
```

Dot File: "ext_ident_bool_1789_arg0.c"

An example of a dot graph associated to a C source file.



Magic labels

The possibility to add labels that would be printed in the C source file was also added in SBV and in Copilot.

The prover was not able to prove long expressions (some were 500000 characters long).

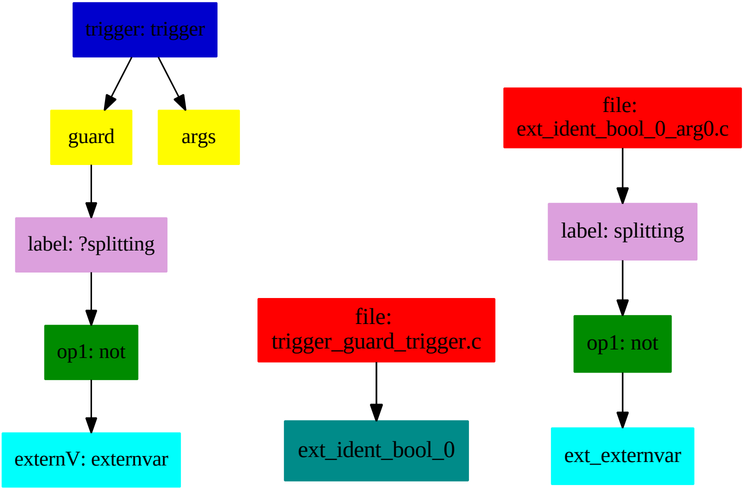
So we need to add a special instruction that has one only role : split the AST into smaller ones that can be easily provable. This instruction has to be totally useless regarding to the semantics of the language. Labels do not change the semantics of the program. So why not using them ?

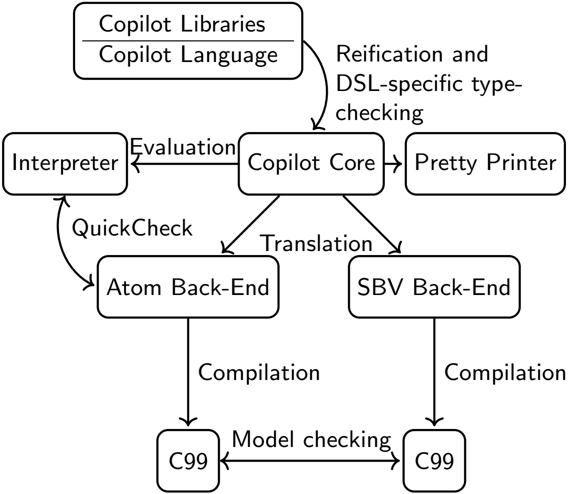
The idea is to call the function identity when we encounter a magic label. This is equivalent to the transformation : $e \to_\beta (\lambda x.x)e$.

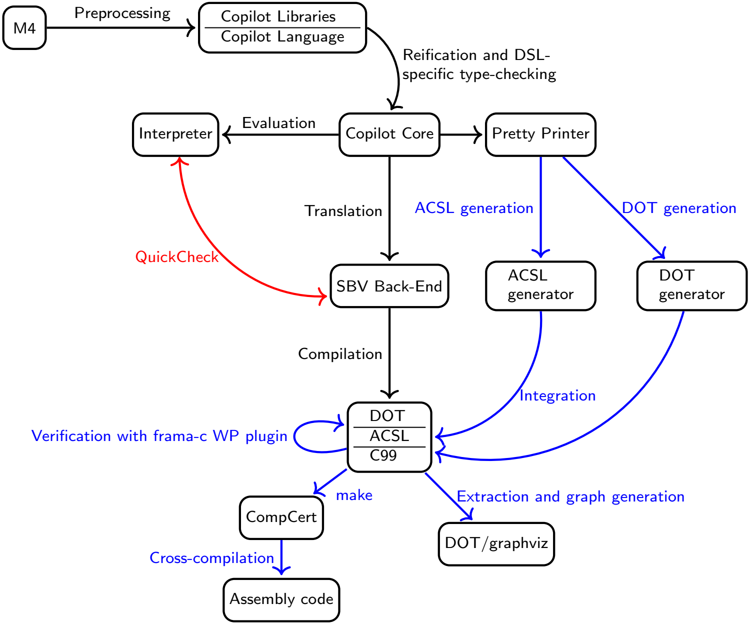
Magic labels: example

```
import qualified Copilot.Compile.SBV as S

alt :: Stream Bool
alt = (label "?splitting" $ not $
      externB "externvar" Nothing)

spec :: Spec
spec = do
trigger "trigger" (alt) []

main = do
reify spec >>= S.proofACSL S.defaultParams
```

Magic labels: example




Copilot Libraries
Copilot Language
Reification and DSL-specific type-checking
Interpreter
Evaluation
Copilot Core
Pretty Printer
QuickCheck
Translation
Atom Back-End
SBV Back-End
Compilation
Compilation
C99
Model checking
C99


M4
Preprocessing
Copilot Libraries
Copilot Language
Reification and DSL-specific type-checking
Interpreter
Evaluation
Copilot Core
Pretty Printer
Translation
ACSL generation
DOT generation
QuickCheck
SBV Back-End
ACSL generator
DOT generator
Compilation
Integration
Verification with frama-c WP plugin
DOT
ACSL
C99
make
Extraction and graph generation
CompCert
DOT/graphviz
Cross-compilation
Assembly code

1. Preliminaries
2. Working on the backend
   - Hand-written ACSL
   - ACSL generation
   - Bug finding
3. Applications

And it works !

Some front-end bugs :

- A reduction of 2^x to $2 << x$ instead of $1 << x$.
- A reduction of 0^0 to 0 instead of 1.
- A reduction of 0^x to 0 instead of $mux(x == 0)(1)(0)$.

Backend bug

A major SBV back-end bug (never detected by model checking).
In a 900 characters long contract of a 100 lines of C code file :

```
/*@
ensure s27 ==
  ((ext_sqrt_0) / (ext_max_time_for_hor_violation));
*/
SBool trigger(...)
{
  const SDouble s11 = ext_max_time_for_hor_violation;
  const SDouble s13 = ext_sqrt_1;
  const SDouble s27 = s13 / s11;
}
```

Backend bug

This similar bug can be generated with the following haskell code :

```
x = externFun "f" [arg 0]
y = externFun "f" [arg 1]
s :: Stream Double
s = x + (y + x)
```

```
/*@
ensure \result ==
    ((ext_f_0) + ((ext_f_1) + (ext_f_0)));
*/
SBool trigger(...)
{
  const SDouble s0 = ext_f_0;
  const SDouble s1 = ext_f_1;
  const SDouble s2 = s1 + s1;  // should be s1 + s0;
  const SDouble s3 = s0 + s2;
  return s3;
}
```

Self-separation criterion

We use the criterion defined in *State-Based Implicit Coordination and Applications* by Anthony J. Narkawicz and César A. Muñoz. The implementation is 262 lines long, generating in prover mode 433 source files, that are verified by frama-c in 2 min and 39 sec on an intel i5-4200U using the following bash command (which uses GNU parallel) :

```
parallel frama-c -wp -wp-out . -wp-timeout 20
-wp-prover CVC4 -wp-split {} ::: *.c | tee
>logfwp >(grep 'Proved\|Unknown\|Timeout\|Failed
\|Qed:\s\|CVC4:\s\|Parsing .*\.c' > logfwpcompact)
>(grep 'Proved\|Qed:\s\|CVC4:\s\|Unknown\|Timeout
\|Failed\|Parsing .*\.c')
```

In compiling mode, the copilot toolchain generates only 19 files, which are compiled in less than a second with CompCert.

1. Preliminaries
2. Working on the backend
3. Applications
   - Self-separation criteria
   - TCAS II
   - Well-Clear

TCAS II

We use the TCAS II implementation in PVS and detailed in *A TCAS-II Resolution Advisory Detection Algorithm* by César A. Muñoz, Anthony J. Narkawicz and James Chamberlain.

We implemented the alert trigger, and the corrective trigger. Both are 447 lines long.

Alert only version :

- Proof mode : 150 files verified in 13 min and 8 sec
- Compile mode : 4 files compiled in 2 sec

Corrective trigger version :

- Proof mode : 1790 files verified in
- Compile mode :

1. Preliminaries
2. Working on the backend
3. Applications
   - Self-separation criteria
   - TCAS II
   - Well-Clear

Well-Clear

Last implementation : Well-Clear criterion defined in *A Family of Well-Clear Boundary Models for the Integration of UAS in the NAS* by César A. Muñoz, Anthony J. Narkawicz and James Chamberlain, María Consiglio and Jason Upchurch.

Questions

Questions ?