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

Georges-Axel Jaloyan

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

August 24, 2015

1. Preliminaries
   - Copilot language
   - ACSL
   - Copilot toolchain
2. Conclusion

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,
--    /!\ 12935301461158671551,...}
```

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. Conclusion

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:

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

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):

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

1. Preliminaries
   - Copilot language
   - ACSL
   - Copilot toolchain
2. Conclusion

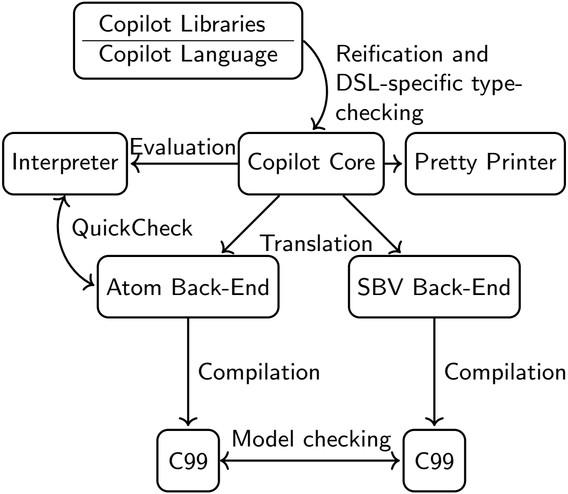

Figure: The Copilot toolchain[3]

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

Questions

Questions ?