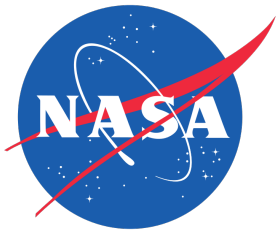
Undergraduate internship report

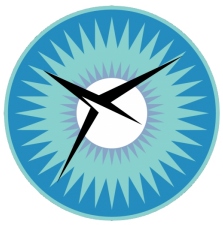
Copilot, traceability of an EDSL generated code.

Author :
Georges-Axel Jaloyan

Supervisor :
Dr. Alwyn E. Goodloe



NASA Langley Research Center



National Institute of Aerospace

École Normale Supérieure

28 August 2015

Introduction

This report is referring to an internship from the 8th of June 2015 to the 14th of August 2015 at *NASA Langley Research Center* also abbreviated as *LaRC*. According to the security policy of the NASA administration, my office was located at the *National Institute of Aerospace* (or *NIA*), a non-profit institute specialized in aerospace and atmospheric research[1].

The main goal of the internship was to work with Copilot, a tool for writing monitors of ultra-critical embedded systems. I had more specifically to work on the output monitor in C and manage writing ACSL contracts automatically in them, and to a broader extent, ensure some traceability of the compilation process, from the Haskell source code, down to the binary, in order to get Copilot to the practical level, and lay the bases for a DO-178B certification.

Contents

[1] http://www.nianet.org/

1 Preliminaries

1.1 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 [10]. Those Copilot programs, can either be compiled to C using two back-ends : SBV or ATOM, interpreted, or analyzed using static analysis tools (CBMC, Kind).

1.1.1 Syntax

Copilot Language uses streams (infinite lazy lists in Haskell) that can be either external streams (obtained by sampling an external variable periodically), or internal streams 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, ..}
```

Each operator and constant has been lifted to Streams (working pointwise), so that it is possible to simply do `1 + nats` instead of `zipWith (+) (repeat 1) nats`, which simplifies the writing of such monitors.

There are two more operators working on Streams : (++) which prepends a finite list to a Stream, and drop, which drops a finite number of elements at the beginning of a Stream. Given that a monitor cannot predict the future, it is impossible to drop elements from a Stream that has no elements prepended using (++).

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

Here is a classical example of the Fibonacci sequence.

```
fib :: Stream Word64
fib = [1,1] ++ (fib + drop 1 fib)
```

It is also possible to cast one Stream to another. For this purpose, two different operators are provided. The `cast` operator can cast safely from one stream to another (in which there is no risk of overflow). The `unsafeCast` operator is used to do almost all casts (except from float to int).

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

1.1.2 Interaction

A copilot monitor can interact with the target program in two ways. First, it can sample external variables, which become Streams. For example, a variable named "x", defined in the target program using `unsigned char x;` can be sampled in a Copilot program by the keyword `extern a b`, where a is the name of the variable, and b is a context for the interpreter (which can be Nothing).

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

Which gives something like (the types have to written explicitly, given that it is not possible to infer the type a an extern) :

```
x :: Stream Word8
x = extern "x" (Just [0,0..])

x2 = externW8 "x" Nothing
```

Similarly, it is possible to sample arrays (only one value at a time on fixed size array, using a Stream of indexes), functions (Stream of arguments), and structures.

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

Finally, a Copilot program can have effects on the target program, using triggers or observers, which can be called by the corresponding operators :

```
trigger :: String -> Stream Bool -> [TriggerArg] -> Spec
observer :: Typed a => String -> Stream a -> Spec
```

A more detailed example named `over_temp_rise` is given in the Copilot manual [12].

For the purpose of the study, the author first wrote a grammar and typing rules for Copilot (which can be found in the Copilot manual [9]). The difficulty resides in the fact that Copilot is an EDSL, thus its lexing, parsing and typing are done using Haskell's. A particular attention has been paid to make the grammar and the typing rules self sufficient, even by restriction some idiomatic constructs, making them unspecified. For instance, strings are only limited to basic one block declarations, lists or arguments are reduced to inductive constructs using explicit type constructors, ...

1.2 Introduction to ACSL and frama-c

ACSL is a language in which we can write specifications for a C program. 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; }
```

An easy way to check that the contract holds is to use Floyd-Hoare logic to verify that the program corresponds to the specification. A Floyd-Hoare triple is in the form $\{P\}\ prog\ \{Q\}$ where *prog* is a program fragment, P and Q are *logical assertions* over program variables. P is called the precondition and Q the postcondition, and writing that $\{P\}\ prog\ \{Q\}$ holds is equivalent to saying that if P holds, then after the execution of *prog*, Q holds (unless *prog* encounters an error, or does not terminate). Here is an example of a proof tree of a program [2]:

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

1.3 Copilot toolchain

Copilot is first reified using standard observation techniques[2]. In its reified form, it can either be interpreted, printed or compiled, using two different back-ends, the first one using Atom, the second one SBV. Both compilers generate C-code, which can be afterwards compared to each other, to ensure that both are semantically equivalent, using model checking techniques. The Figure 1 shows a global view of the different tools used in Copilot.

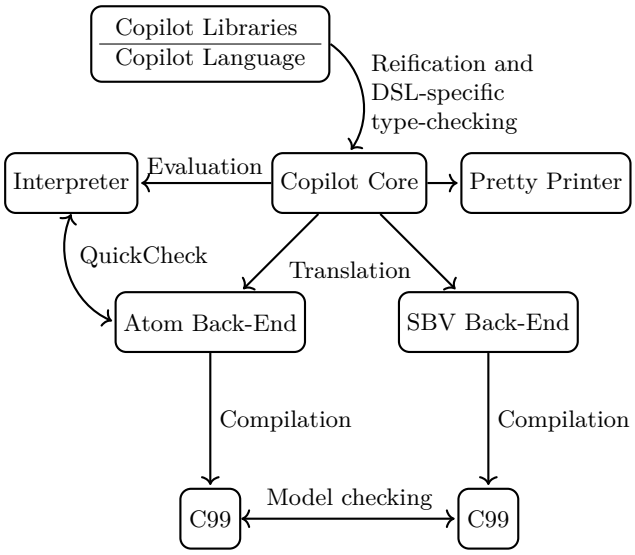


Figure 1: The Copilot toolchain [11]

1.4 Main objective and first modifications

The main goal was to write ACSL contracts in the code produced in order to have a high degree of confidence in the compiler, without having to prove the compiler, which is too complicated given that we use an EDSL. For this, we had to modify the backends, and the libraries producing the code (namely SBV and ATOM). A first look in the C code produced showed that given its structure, the code produced by ATOM would be really hard to prove (many hacks, non idiomatic expressions, bad splitting into functions). Henceforth, we decided to focus only on SBV, trying to produce the contracts in the Copilot backend for SBV.

2 First attempt

2.1 Hand-written ACSL

To get acquainted with ACSL and the code generated, the author first wrote some ACSL contracts for C files in order to look at what subset of the ACSL language would be needed, and what part of ACSL was implemented in the plugins.

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

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

[2]These techniques can be found in [1]

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

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

After compiling using the SBV backend, we generate several files, one of which is called `update_state2.c`. We manually write an ACSL contract for the function present in it, and we run several tests, playing with some parameters.

```
/*@
requires ptr_2 < 0x000078;
requires \valid(queue_2 + (0..0x0002U-1));
assigns \nothing;

ensures \result == ( queue_2[ptr_2 % 0x0002U] && queue_2[(ptr_2 + 0x0001U) %
 0x0002U]);
*/
SBool update_state_2(const SBool *queue_2, const SWord16 ptr_2)
{
const SBool   s0 = queue_2[0];
const SBool   s1 = queue_2[1];
const SWord16 s2 = ptr_2;
const SBool table0[] = {
s0, s1
};
const SWord16 s4 = (0x0002U == 0) ? s2 : (s2 % 0x0002U);
const SBool   s5 = table0[s4];
const SWord16 s7 = s2 + 0x0001U;
const SWord16 s8 = (0x0002U == 0) ? s7 : (s7 % 0x0002U);
const SBool   s9 = table0[s8];
const SBool   s10 = s5 && s9;

return s10;
}
```

By running `frama-c -wp -wp-out . -wp-prover PROVER` with several provers. First we tried on the original file with the contracts. Then we changed also the laziness of the logical operators (both in the contract and the code), leading to an impossibility in the proof (Timeout for Z3).

This means that no prover can prove a contract with bitwise boolean operators inside. That's why the SBV library had to be modified so that when having logical operators on bool values, those would be compiled to lazy variants, given that such an approximation is safe (which is obviously not the case on non boolean values)[3]. An other interesting observation is that if we delete the guard `requires ptr_2 < 0x000078;`, then the prover Z3 concludes that the code is clearly unsafe, which can create problems given that the functions are spread on several files and no global invariant is available.

[3]https://github.com/LeventErkok/sbv/issues/177

Original file.

```
[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.

```
[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
```

No guard version.

```
[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
```

2.2 Automatic generation of contracts by induction on the syntax.

Thus, we can see that the SBV backend generates one function per expression of the reified AST (abstract syntax tree). Thus the main task consists in generating a contract for the expression, which can be done by induction on the syntax. This is very similar to writing a pretty printer for expressions (corresponding to the token $\langle funStream \rangle$ in the grammar), except for some tricks, which have to be taken into account, which are often due to the absence of implementation of some features described in ACSL.

- No let bindings can be written in ACSL contracts.

  It was decided to do not support let bindings during ACSL contract generation, thus making a code containing let bindings unprovable. Nevertheless, the programmer can use a functional style, changing the let bindings into function, at the cost of having arguments in order to use variables that should be in scope for let bindings.

- No mathematical functions are available (sin, cos, tan, sqrt, ...).

  This is quite normal, given that on embedded systems, the access to math.h is not guaranteed. Thus we decided to transform each trigonometrical function into a call to an external function, called the same way (or sinf if working on floats), and it's up to the user to provide an implementation of the function (or to choose the standard implementation in math.h).

- No global invariants are implemented in WP plugin.

  We split the dereferencing of the pointer in an external function (called ident, which is equivalent to the identity), which allows to have simple enough contracts to be able to prove the assertions, and given that the splitting does not change the semantics of the program (it is similar to an beta-expansion).

- No bitwise operator.

  SBV library was changed in that purpose, and bitwise operators have been deprecated[4].

2.3 Values Analysis vs Weakest Liberal Precondition.

The value analysis plugin was also tested given that it supports global invariants. The problems resides in the facts that it can prove the safety of one step of sampling, and when trying to generalize the proof by adding an infinite loop of the step function, all the values are evaluated to Top using widening. This causes the analysis to fail (almost all instructions become unsafe). A solution the author implemented consists in telling the analyzer to unroll the infinite loop[5], which allows the analyzer to run on a finite number of loop steps, which is enough to prove an assertion of the type : until the step x, this monitor is safe, which can be enough for a plane or a car (a civilian plane is not supposed to fly more than two days), but is clearly insufficient for the space industry.

For these reasons, it was decided to focus on the WP plugin, even if we added a feature to try the value-analysis plugin.

[4] https://github.com/LeventErkok/sbv/issues/177

[5] Using the option `-slevel n` where `n` is the number of steps

2.4 CompCert.

After managing to write the generation of contracts, we finally changed the compiler to CompCert, which generates semantically equivalent assembly code in a prover manner (the option make all in the makefile compiles every source file in a object file and assembles all the objects into an archive using a standard archiver). For this purpose, SBV library also needed to be changed[6].

2.5 Bug finding

After all the first steps done, the author did some bug tracking in the Copilot toolchain, looking for front-end errors, and even some back-end errors. Indeed, front-end errors are a common issue for high insurance compilers, which can not be avoided even by proving methods [14].

First, a bug in the code generation using file handles was present in the source code, resulting in blank files randomly generated in SBV. This was corrected by deleting the handles, and going at a higher level construct with the functions writeFile of the haskell standard library[7].

As expected front-end errors were present in the source code. Those were mainly mathematical front-end reduction for syntactic sugar which were false.

- A reduction of 2^x to $2 << x$ instead of $1 << x$[8].
- A reduction of 0^0 to 0 instead of 1[9].
- A reduction of 0^x to 0 instead of $mux(x == 0)(1)(0)$[10].

What was more surprising is that a serious back-end bug was present in the translator to SBV (also called the SBV backend), which was not present in the ATOM backend, and which was never detected by model checking techniques (given that the code generating the bug was never tested). The bug was found by manually looking in a C source file of 100 lines, which contained a contract 900 characters which evaluated to Unknown status using WP plugin. By extracting the interesting piece of code, we had :

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

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)
```

Which will generate that following C code.

```
/*@
ensure \result == ((ext_f_0) + ((ext_f_1) + (ext_f_0)));
```

[6] https://github.com/LeventErkok/sbv/issues/175#issuecomment-114105589
[7] https://github.com/Copilot-Language/copilot-sbv/commit/5a7f274be38382155a7dd3422d9003ec349f92c0
[8] https://github.com/Copilot-Language/copilot-language/commit/65eb97fb2000b0a0db512e873b8f9ce82b2ed68b
[9] https://github.com/Copilot-Language/copilot-language/commit/65eb97fb2000b0a0db512e873b8f9ce82b2ed68b
[10] https://github.com/Copilot-Language/copilot-language/commit/577208958ee48355bbc5d7ce7ed5430130f00c9a

```
*/
SBool trigger(...)
{
const SDouble s0 = ext_f_0;
const SDouble s1 = ext_f_1;
const SDouble s2 = s1 + s1;  // Here is the error. It should be s1 + s0;
const SDouble s3 = s0 + s2;
return s3;
}
```

This was due to the fact that when we use an external function several times in one stream, the sbv backend just forgets to take into account the number of times that the external function has already been called with other arguments, so it always gets the value from the last call of the function. This was corrected by changing a key in a map[11].

2.6 Ensuring traceability

As requested by the DO-178B norm regarding the safety of critical flight avionics, a traceability has to be ensured between the origin haskell source code file and the generated C code produced. For that purpose, the author implemented a labeling system, similar to the trace instruction from Debug.Trace, but prints its label in the C source file instead of stdout. Moreover, the fact that the expressions could be huge after reification (all functions are beta expanded in expressions), it caused problems with frama-c which could not parse contracts of 500000 characters long.

For this purpose, the author had the idea to implement three features. The first is a prover mode, which will optimize the code in order to make it easier to prove (by splitting absolute values in half for example). The second is a pretty printer of the AST into a dot file, which then generates a graph of what a C file is supposed to do (this graph is equivalent to the ACSL contract, so it is easier to see a problem in the contract by looking at the AST graph). The last are magic characters in labels, which can affect the generated C code when the compilation is in mode prover. The only one implemented is the character `?` which causes the backend to cut the AST at this node and create an other one. This is done by the method presented in the paragraph 2.2 about the dereferencing of pointers in a box using the identity external function which is shown in the Figure 2 for the following simple Copilot code :

```
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
```

At the end, after all these modifications we had to change the toolchain according to the Figure 3

[11] https://github.com/Copilot-Language/copilot-sbv/commit/6bc043117eac17e918ac7a6363b866058911e59f

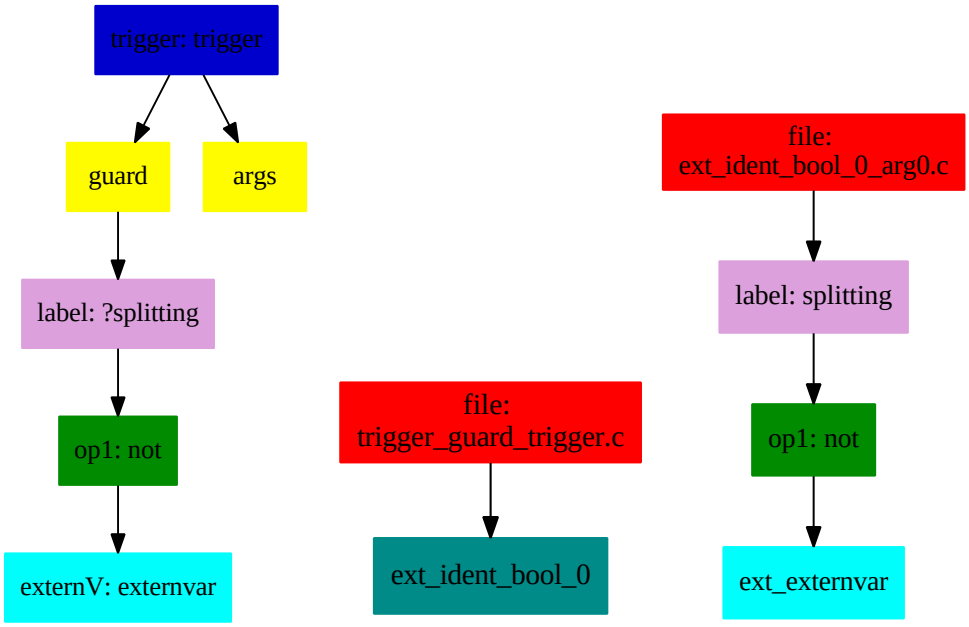


Figure 2: The splitting process. The AST on the left is split into two smaller ASTs, that then generate two separate functions into two separate files, making it easier to prove the contracts for them.

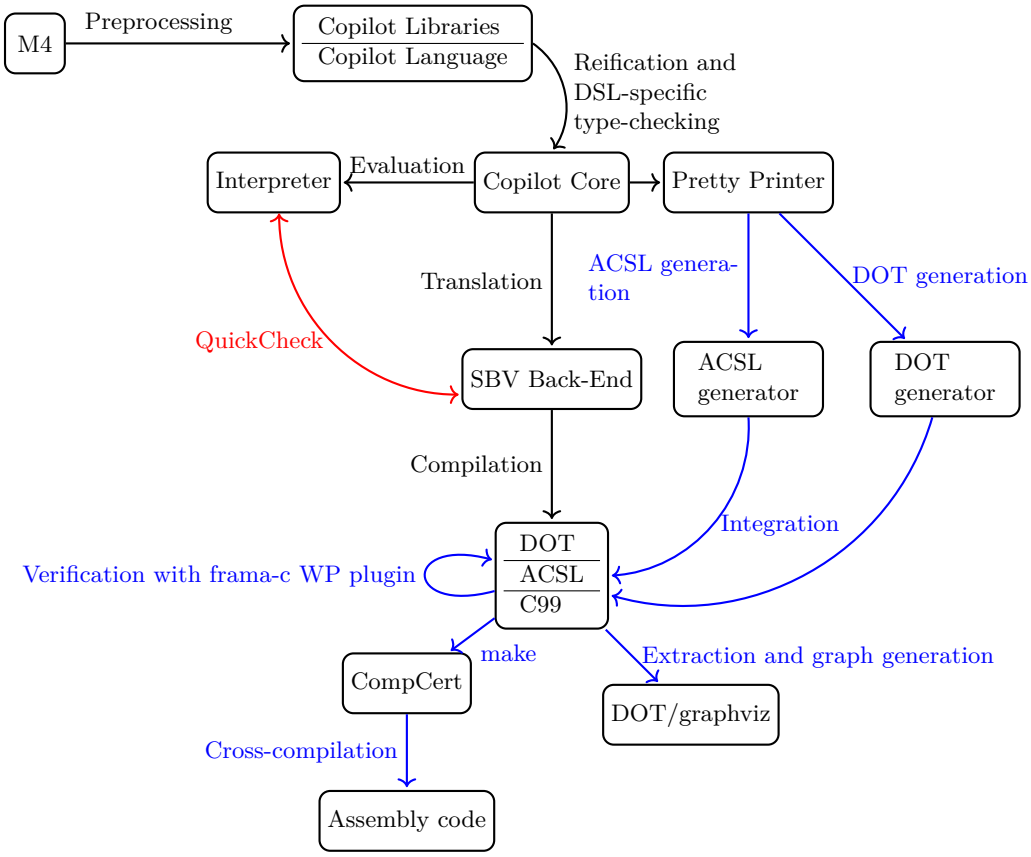


Figure 3: The new Copilot toolchain. The red arrows are the one to implement in the future. The blue ones are the ones implemented by the author.

3 Practical steps

3.1 First application : Self-separation criteria

The first monitor written in Copilot is an application to the conflict resolution and self-separation recovery criteria elaborated by Dr. César A. Muñoz.

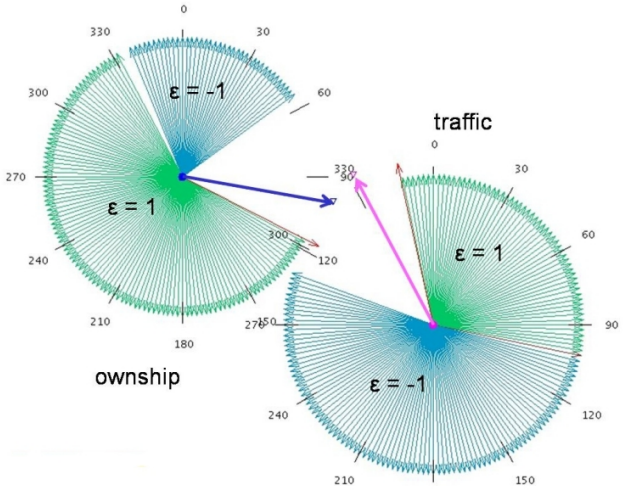


Figure 4: Visualization of Horizontal Criterion for Conflict Avoidance from [7].

The first criterion about conflict resolution (or avoidance) guarantees that even if two planes (one called the *ownship* and the other the *intruder*) have different control algorithm, it is possible to guarantee that they will maintain a safe separation which usually corresponds to a 5 nm horizontal separation and 1000 ft vertically. This gives two sub-criterion, a horizontal (shown in Figure 4) and a vertical one. Which can then be assembled into a bigger 3 dimension criterion.

The second criterion about self-separation recovery guarantees that when two planes are in a situation of violation of their safe-separation, the algorithms will be able to recover their safe-separation within t seconds.

The details could be found in [3] and [6], and here are their inputs and outputs :

- Inputs : Velocity, Position of the ownship and the intruder (both 3 dimensional vectors). The output given by the algorithm of the ownship. The minimum horizontal and vertical separations D and H (usually $D = 5nm$, $H = 1000ft$). The direction parameters for horizontal and vertical resolution, which are a value that can either be + or - 1 ; the two planes share the same value. The max time for violation t.
- Outputs : two boolean values, corresponding each other to the violation of a criterion.

The implementation is 262 lines long[12]. The proof mode generates 433 C source files, which are then verified by frama-c in 2 minutes and 39 sec on an intel i5-4200U[13][14] 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')
```

[12]https://github.com/Copilot-Language/examplesForACSL/blob/master/example29/main.hs
[13]http://ark.intel.com/fr/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz
[14]The whole process is logged in https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/2bead489ca551eb1c54aa7fa4408697e07344f26/example29/copilot-sbv-codegen/logfwp and a compact log better suited for readability is in https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/2bead489ca551eb1c54aa7fa4408697e07344f26/example29/copilot-sbv-codegen/logfwpcompact

In compiling mode, the copilot toolchain generates only 19 C source files, which are compiled in less than a second with CompCert[15].

3.2 Second application : TCAS II

The second monitor written in Copilot is an implementation of a simplified TA for TCAS II (Traffic alert and Collision Avoidance System). The TCAS II is a system present on all civilian planes that is composed of two parts : the *Traffic Advisory* (TA) which triggers an audible alert if two planes are too close from each other, and a *Resolution Advisory* (RA) which sends instructions to the pilot in order to avoid collision and recover a safe separation between the two planes.

The TA starts emitting alerts if the intruder plane enters in a safe cylinder (called the TA region) of typically 3.3 nautical miles (nm), which corresponds to 40 seconds before collision. The RA is triggered if the conflicts occurs in the RA region, which typically corresponds to 2.1 nm (or 25 sec). The RA is only giving advice about vertical speeds, and not on heading, as shown in the Figure 5.

Figure 5: The TCAS II resolution advice about vertical speed. The orange zone corresponds to the set of unsafe vertical speeds. The pilot has to maneuver so that its vertical speed gets out of this area. The intruder traffic is shown by the orange square, and the RA given to the other plane is represented by the arrow next to it (here the other traffic also has to descend, as the ownship). The two digit number shown is the intruder's altitude relative to the ownship (in hundreds of feet).

A typical violation starts with an intruder plane entering the TA region, which triggers the audible alert "traffic, traffic". If the violation continues until the RA region, this time the system will evaluate if the resolution advice will be corrective or preventive. The preventive resolution advice, consist in giving indications to maintain altitude or vertical speed to the pilot, so that the collision will not happen. The corrective (and hence the most dangerous), gives indication that have to be followed in order to avoid an imminent collision. In this case an other audible alert is emitted, which either asks to "descend, descend", or "climb, climb NOW", or to "Adjust Vertical Speed, Adjust". When the conflict is definitely avoided, the TCAS sends a message saying that the situation is "clear of conflict" [16] [17].

More specifically, the author implemented two of these features : the TCAS II RA alert, and the discrimination function between the preventive and corrective advice. Those were extracted from the PVS model of the NASA library [18], which can also be found in [4].

[15] https://github.com/Copilot-Language/examplesForACSL/tree/db0a348a8a6e3d33b0046d2b9b06c938705f0b32/example29/copilot-sbv-codegen

[16] https://www.youtube.com/watch?v=z-6zF9PEtdU

[17] https://www.youtube.com/watch?v=OrYqIU0NxHQ

[18] https://github.com/nasa/pvslib/tree/master/TCASII

The implementation is 447 lines long[19]. The proof mode generates 1790 C source files, which are then verified by frama-c in 4 hours and 38 min on an intel i5-4200U[20]. In compiling mode, the copilot toolchain generates only 5 C source files, which are compiled in less than two second with CompCert[21].

3.3 Third application : Well-clear criterion

The last application, is the implementation of the Well-Clear Violation criteria (also known as *WCV*) in Copilot defined in [13], [5] and [8] by César A. Muñoz, Anthony J. Narkawicz and James Chamberlain, María Consiglio and Jason Upchurch. Those criteria ensure that two planes, if this criterion is met, are in a safe situation. To some extent, this criterion is a guarantee that, if respected, a TCAS alert would not be triggered. The criterion has an horizontal and vertical part, as shown by its formula :

$$\begin{aligned} WCV_{t_{\text{var}}}(\mathbf{s}, s_z, \mathbf{v}, v_z) \equiv\ & \texttt{Horizontal\_WCV}_{t_{\text{var}}}(\mathbf{s}, \mathbf{v})\ \texttt{and} \\ & \texttt{Vertical\_WCV}(s_z, v_z), \end{aligned}$$

$$\begin{aligned} \texttt{Horizontal\_WCV}_{t_{\text{var}}}(\mathbf{s}, \mathbf{v}) \equiv\ & \|\mathbf{s}\| \leq \texttt{DTHR or} \\ & (d_{\text{cpa}}(\mathbf{s}, \mathbf{v}) \leq \texttt{DTHR and}\ 0 \leq t_{\text{var}}(\mathbf{s}, \mathbf{v}) \leq \texttt{TTHR}), \\ \texttt{Vertical\_WCV}(s_z, v_z) \equiv\ & |s_z| \leq \texttt{ZTHR or}\ 0 \leq t_{\text{coa}}(s_z, v_z) \leq \texttt{TCOA}. \end{aligned}$$

where t_{coa} and d_{cpa} are defined by the following :

$$d_{cpa}(\mathbf{s}, \mathbf{v}) = \| \mathbf{s} + t_{cpa}(\mathbf{s}, \mathbf{v})\mathbf{v} \|$$

$$t_{\text{coa}}(s_z, v_z) \equiv \begin{cases} -\frac{s_z}{v_z} & \text{if } s_z v_z < 0, \\ -1 & \text{otherwise.} \end{cases}$$

and t_{var} is a time variable which can be τ, $tcpa$, τ_{mod} or tep (τ and τ_{mod} are two time thresholds defined by the TCAS II algorithm) defined in [13].

This allows us to define different areas of violation, which are included in each other according to the inclusion theorem in [13] and illustrated by the figure 6. The implementation is 282 lines long[22]. The proof mode generates 980 C source files, which are then verified by frama-c in 10 minutes and 2 sec on an intel i5-4200U[23]. In compiling mode, the copilot toolchain generates only 14 C source files, which are compiled in two seconds with CompCert[24].

A unit test was written for the generated code in C99, with some test cases provided (not published). All of those scenarii are involving two planes, giving a sampling rate of 1 second, with planes entering and getting out of violation. This was used for writing a monitor that would run on a ground station, which has to monitor both real UAVs and fake simulated vessels (there is no way for the monitor to discriminate both). The general structure of the monitor is in the figure 3.3.

[19] https://github.com/Copilot-Language/examplesForACSL/blob/43beff24ecf6c71aa20e544c9053fd7ed5491fc9/example36/main.hs

[20] The whole process is logged in https://github.com/Copilot-Language/examplesForACSL/blob/3aa55009e69351f9e5ea2d33a5eea067fa2dbafc/example36/copilot-sbv-codegen/logfwp2.tar.gz and a compact log better suited for readability is in https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/3aa55009e69351f9e5ea2d33a5eea067fa2dbafc/example36/copilot-sbv-codegen/logfwpcompact

[21] https://github.com/Copilot-Language/examplesForACSL/tree/39c926a773f6cabbe4698fb13e24a3af9fb3e38f/example36/copilot-sbv-codegen

[22] https://github.com/Copilot-Language/examplesForACSL/blob/9b58e239284368c4e5665231f53b209658cf62b3/WCV/main.hs

[23] The whole process is logged in https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/ef85521e9e5df85368f9f51f686e171b6c1413da/example38/copilot-sbv-codegen/logfwp and a compact log better suited for readability is in https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/ef85521e9e5df85368f9f51f686e171b6c1413da/example38/copilot-sbv-codegen/logfwpcompact

[24] https://github.com/Copilot-Language/examplesForACSL/tree/56e35879cfb21286e924227dd61a770c05a43930/example38/copilot-sbv-codegen

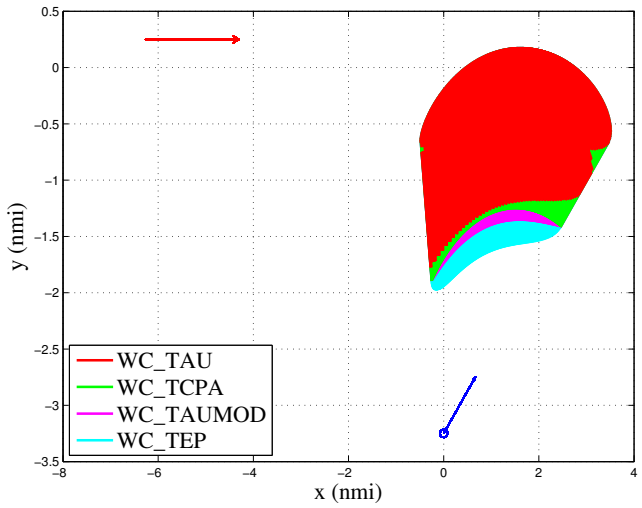


Figure 6: We clearly see here $WCV_\tau \Rightarrow WCV_{tcpa} \Rightarrow WCV_{\tau_{mod}} \Rightarrow WCV_{tep}$. Here we decided to choose $DTHR = 1nmi$, $TTHR = TCOA = 30s$, and $ZTHR = 475ft$.

More specifically, the planes would communicate with a first interface (named GUI here), that would do the conversion between raw data from the planes, and translate them into audible alerts (or visible) and feed the monitor on stdin. The monitor will check if any two of planes trigger a WCV alert, after computation by the driver.c. Given that the data is given into latitude and longitude coordinates, a converter has to be used to convert them in x, y, z coordinates[25]. It is planned to test this monitor in a real situation in the Fall 2015.

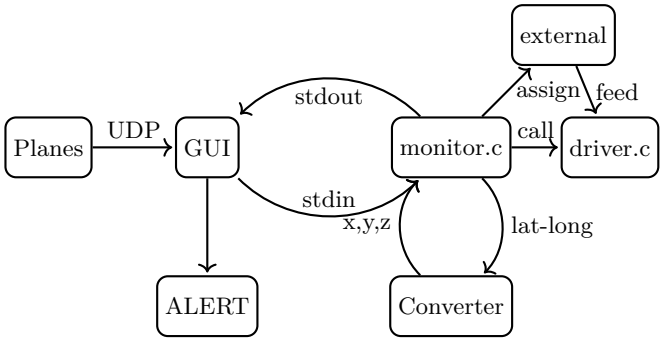


Figure 7: The overall structure of the monitor

Conclusion

All in all, this 10 week internship was really encouraging, with a lot of tools I discovered and used efficiently in order to improve the reliability and traceability of a synchronous EDSL in an original way. I also implemented three examples of programs that Copilot is targeting specifically (runtime monitor for hard real-time distributed systems in C), in order to show that the language is mature enough to start experimentations in real condition, and to show that taking it to an other level is neither pointless, nor groundless (DO-178B certification).

To reach the next step, there is still some work to do, such as code refactoring for Haskell, make the C source code comply to ISO/IEC 9899:1999, do more testing both with unit tests and real situation, add new features (matrix, structs, math functions), and strengthen the whole Haskell code by either intensive unit testing or by extending the ACSL contracts to check more than the simple compilation process.

[25]More precisely, z is kept the same for all planes, x and y are obtained by projecting the ownship and the intruder on a plane tangent to the earth on the middle point of the great circle that connects the two vertical projection points.

References

[1] A. Gill, "Type-safe observable sharing in haskell," in *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell '09. New York, NY, USA: ACM, 2009, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1596638.1596653

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

[3] C. Muñoz, R. Butler, A. Narkawicz, J. Maddalon, and G. Hagen, "A criteria standard for conflict resolution: A vision for guaranteeing the safety of self-separation in NextGen," NASA, Langley Research Center, Hampton VA 23681-2199, USA, Technical Memorandum NASA/TM-2010-216862, October 2010.

[4] C. Muñoz, A. Narkawicz, and J. Chamberlain, "A TCAS-II resolution advisory detection algorithm," in *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2013*, no. AIAA-2013-4622, Boston, Massachusetts, August 2013.

[5] C. Muñoz, A. Narkawicz, J. Chamberlain, M. Consiglio, and J. Upchurch, "A family of well-clear boundary models for the integration of UAS in the NAS," in *Proceedings of the 14th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, no. AIAA-2014-2412, Georgia, Atlanta, USA, June 2014. [Online]. Available: http://arc.aiaa.org/doi/abs/10.2514/6.2014-2412

[6] A. Narkawicz and C. Muñoz, "State-based implicit coordination and applications," NASA, Langley Research Center, Hampton VA 23681-2199, USA, Technical Publication NASA/TP-2011-217067, March 2011.

[7] A. Narkawicz, C. Muñoz, and G. Hagen, "An independent and coordinated criterion for kinematic aircraft maneuvers," in *Proceedings of the 14th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, no. AIAA-2014-2859, Georgia, Atlanta, USA, June 2014. [Online]. Available: http://arc.aiaa.org/doi/abs/10.2514/6.2014-2859

[8] A. J. Narkawicz, C. A. Muñoz, J. M. Upchurch, J. P. Chamberlain, and M. C. Consiglio, "A well-clear volume based on time to entry point," NASA, Langley Research Center, Hampton VA 23681-2199, USA, Technical Memorandum NASA/TM-2014-218155, January 2014.

[9] NASA, "The copilot manual," 2015.

[10] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *Runtime Verification*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Springer Berlin Heidelberg, 2010, vol. 6418, pp. 345–359. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16612-9_26

[11] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Experience report: A do-it-yourself high-assurance compiler," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12. New York, NY, USA: ACM, 2012, pp. 335–340. [Online]. Available: http://doi.acm.org/10.1145/2364527.2364553

[12] ——, "Copilot: Monitoring embedded systems," in *Innovations in Systems and Software Engineering: Special Issue on Software Health Management*, vol. 9, no. 4. Springer, 2013, pp. 235–255, preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/isse.html.

[13] J. M. Upchurch, C. A. Muñoz, A. J. Narkawicz, J. P. Chamberlain, and M. C. Consiglio, "Analysis of well-clear boundary models for the integration of UAS in the NAS," NASA, Langley Research Center, Hampton VA 23681-2199, USA, Technical Memorandum NASA/TM-2014-218280, June 2014.

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

Acknowledgment

The author would like to thank Dr. Alwyn Goodloe for his supervising, Dr. Lee Pike, Dr. Levent Erkok, Dr. César A. Muñoz, Dr. Patrick Quach, Dr. Andrew Smith, Dr. Natasha Neogi, Dr. Mariano Moscato, Dr. Jean-Christophe Filliâtre.

Appendix

Simplified grammar for Copilot language.

$\langle defs \rangle$ ::= $(\langle def \rangle)^*$

$\langle ctype \rangle$ ::= `Bool`
| `Int8`
| `Int16`
| `Int32`
| `Int64`
| `Word8`
| `Word16`
| `Word32`
| `Word64`
| `Float`
| `Double`

$\langle def \rangle$ ::= $(\langle id \rangle$`::Stream` $\langle ctype \rangle)?\langle id \rangle$`=`$\langle spec \rangle$

$\langle id \rangle$ ::= `(a-z)(a-z|A-Z|0-9|_|-|')*`

$\langle string \rangle$ ::= `"(a-z|A-Z)(a-z|A-Z|_|0-9)`$^*_{\leq 30}$`"`

$\langle stream \rangle$ ::= $\langle valueList \rangle$`++`$\langle stream \rangle$
| $\langle funStream \rangle$

$\langle valueList \rangle$ ::= $[(\langle vbool \rangle)^*_,]$
| $[(\langle vint \rangle)^*_,]$
| $[(\langle vfloat \rangle)^*_,]$

$\langle vbool \rangle$ ::= `true`
| `false`

$\langle vint \rangle$ ::= `[+|-](0-9)`$^+$

$\langle vfloat \rangle$ ::= $\langle vint \rangle$`.(0-9)`$^+$

$\langle funStream \rangle$ ::= $\langle externV \rangle$ $\langle string \rangle$ $\langle contextV \rangle$
| `label` $\langle string \rangle$ $\langle funStream \rangle$
| `externFun` $\langle string \rangle$ $\langle argList \rangle$ $\langle contextF \rangle$
| $\langle externA \rangle$ $\langle string \rangle$ $\langle stream \rangle$ $\langle int \rangle$ $\langle contextA \rangle$
| $\langle op1 \rangle$ $\langle funStream \rangle$
| $\langle funStream \rangle$ $\langle op2Infix \rangle$ $\langle funStream \rangle$
| $\langle op3 \rangle$ $\langle funStream \rangle$ $\langle funStream \rangle$ $\langle funStream \rangle$
| $\langle dropStream \rangle$

$\langle externV \rangle$ ::= `extern`

$\langle externA \rangle$ ::= `externArray`

$\langle contextV \rangle$ ::= `Nothing`
| `Just` $\langle stream \rangle$

$\langle contextF \rangle$ ::= `Nothing`
| `Just` $\langle stream \rangle$

$\langle contextA \rangle$ ::= `Nothing`
| `Just` $[(\langle valueList \rangle)^*_,]$

$\langle argList \rangle$ ::= $[$`(Arg` $\langle stream \rangle)^*_,]$
| `[]`
| `(Arg` $\langle stream \rangle$`):`$\langle argList \rangle$

$\langle dropStream \rangle$ ::= $\langle id \rangle$
| `constant` $\langle value \rangle$
| `drop` $\langle int \rangle$ $\langle stream \rangle$

$\langle op1 \rangle$ ::= `not` | `abs` | `signum`
| `cast` | `unsafeCast`

$\langle op2Infix \rangle$ ::= `+` | `-` | `*` | `` `mod` `` | `` `div` `` | `/`
| `<` | `<=` | `==` | `/=` | `>=` | `>`
| `||` | `&&` | `` `xor` `` | `==>`

$\langle op3 \rangle$ ::= `mux`

Simplified typing rules for Copilot language.

$$\frac{s \text{ is a } \langle string \rangle \text{ token}}{\Gamma \vdash s : \texttt{String}} \text{ (stringConst)} \qquad \frac{\Gamma \vdash ls : [a] \quad \Gamma \vdash s : \text{Spec } a}{\Gamma \vdash ls +\!+ s : \text{Spec } a} \text{ (Append)}$$

$$\frac{\tau \in \text{inst}(\Gamma(s))}{\Gamma \vdash s : \tau} \text{ (inst)} \qquad \frac{\Gamma \vdash s : \texttt{String} \quad \Gamma \vdash x : \texttt{Stream } \tau}{\Gamma \vdash \texttt{label } s\ x : \texttt{Stream } \tau} \text{ (label)}$$

$$\frac{\Gamma \vdash s : \texttt{Stream } \tau}{\Gamma \vdash \texttt{Arg } s : \texttt{Arg}} \text{ (arg)} \qquad \frac{\Gamma \vdash s : \texttt{String} \quad \Gamma \vdash x : \texttt{Stream } \tau}{\Gamma \vdash \texttt{extern } s\ x : \texttt{Stream } \tau} \text{ (ext)}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{constant } x : \texttt{Stream } \tau} \text{ (valConst)}$$

$$\frac{\Gamma \vdash i : \texttt{Integer} \quad \Gamma \vdash x : \texttt{Stream } \tau}{\Gamma \vdash \texttt{drop } i\ x : \texttt{Stream } \tau} \text{ (drop)} \qquad \frac{\Gamma \vdash s : \texttt{String} \quad \Gamma \vdash a : \texttt{[Arg]} \quad \Gamma \vdash x : \texttt{Stream } \tau}{\Gamma \vdash \texttt{externFun } s\ a\ x : \texttt{Stream } \tau} \text{ (extFun)}$$

$$\frac{\Gamma \vdash s : \texttt{String} \quad \Gamma \vdash i : \texttt{Integral } \tau_1 \Rightarrow \texttt{Stream } \tau_1 \quad \Gamma \vdash m : \texttt{Integer} \quad \Gamma \vdash x : \texttt{[[}\tau\texttt{]]}}{\Gamma \vdash \texttt{externArray } s\ i\ m\ x : \texttt{Stream } \tau} \text{ (extA)}$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \quad op \in \{\texttt{not}\}}{\Gamma \vdash op\ x : \texttt{Stream Bool}} \text{ (op1Bool)}$$

$$\frac{\Gamma \vdash x : \texttt{Integral } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{\texttt{abs}, \texttt{signum}\}}{\Gamma \vdash op\ x : \texttt{Stream } \tau} \text{ (op1Num)}$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \quad \Gamma \vdash y : \texttt{Stream Bool} \quad op \in \{\texttt{||}, \texttt{\&\&}, \texttt{`xor`}, \texttt{==>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream Bool}} \text{ (op2Bool)}$$

$$\frac{\Gamma \vdash x : \texttt{Integral } \tau \Rightarrow \texttt{Stream } \tau \quad \Gamma \vdash y : \texttt{Integral } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{\texttt{`mod`}, \texttt{`div`}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream } \tau} \text{ (op2Integral)}$$

$$\frac{\Gamma \vdash x : \texttt{Fractionnal } \tau \Rightarrow \texttt{Stream } \tau \quad \Gamma \vdash y : \texttt{Fractionnal } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{/\}}{\Gamma \vdash op\ x\ y : \texttt{Stream } \tau} \text{ (op2Fractionnal)}$$

$$\frac{\Gamma \vdash x : \texttt{Num } \tau \Rightarrow \texttt{Stream } \tau \quad \Gamma \vdash y : \texttt{Num } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream } \tau} \text{ (op2Num)}$$

$$\frac{\Gamma \vdash x : \texttt{Eq } \tau \Rightarrow \texttt{Stream } \tau \quad \Gamma \vdash y : \texttt{Eq } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{\texttt{==}, \texttt{/=}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream } \tau} \text{ (op2Eq)}$$

$$\frac{\Gamma \vdash x : \texttt{Ord } \tau \Rightarrow \texttt{Stream } \tau \quad \Gamma \vdash y : \texttt{Ord } \tau \Rightarrow \texttt{Stream } \tau \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream } \tau} \text{ (op2Ord)}$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \quad \Gamma \vdash y : \texttt{Stream } \tau \quad \Gamma \vdash z : \texttt{Stream } \tau \quad op \in \{\texttt{mux}\}}{\Gamma \vdash op\ x\ y\ z : \texttt{Stream } \tau} \text{ (op3)}$$