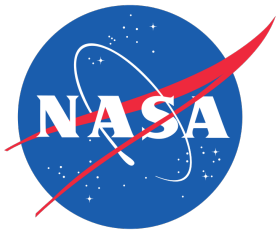
Undergraduate internship report

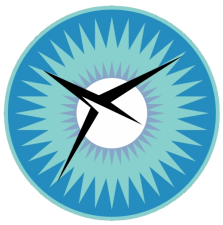
Copilot, traceability of an EDSL generated code.

Author :
Georges-Axel Jaloyan

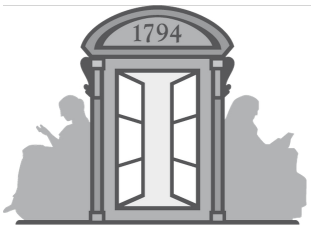
Supervisor :
Dr. Alwyn E. Goodloe



NASA Langley Research Center



National Institute of Aerospace



École Normale Supérieure

28 August 2015

Contents

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*.

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 [4]. 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, but only in a safe way (in which there is no risk of overflow), with the operator `cast`.

```
cast :: (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, which can be called by the trigger operator :

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

A more detailed example named `over_temp_rise` is given in the Copilot manual [6].

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 [3]). 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]:

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

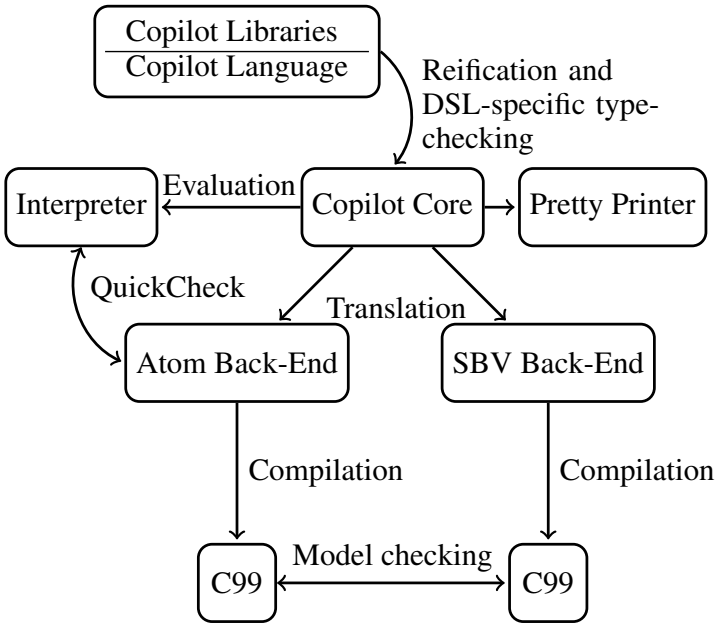


Figure 1: The Copilot toolchain [5]

1.3 Copilot toolchain

Copilot is first reified using standard observation techniques[1]. In its reified form, it can either be interpreted, printed or compiler, 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[2]. The Figure 1 shows a global view of the different tools used in Copilot.

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
```

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

```
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
```

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.

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

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

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

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 [7].

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.

[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

```
/*@
ensure result == ((ext_sqrt_0) + ((ext_sqrt_1) + (ext_sqrt_0)));
*/
bool 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].

3 Practical steps

3.1 First application : Self-separation criterion

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

3.2 Second application : Well-clear criterions

3.3 Third application : TCAS II

Conclusion

The conclusion goes here. you lost

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

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] NASA, "The copilot manual," 2015.

[4] 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

[5] 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

[6] ——, "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.

[7] 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 precious supervising, Dr. Lee Pike, Dr. Levent Erkok for his work on SBV and improves for the needs of the project, Dr. César A. Muñoz for his criterions of Well-Clear and Safe separation, Dr. Andrew Smith, Dr. Natasha Neogi, Dr. Mariano Moscato, Dr. Jean-Christophe Filliâtre.

Appendix