The National Institute of Aerospace / Galois Inc. / NASA LaRC

An Introduction to Copilot

Nis N. Wegmann
niswegmann@gmail.com

Lee Pike
leepike@galois.com

Sebastian Niller
sebastian.niller@nianet.org

Hampton, Virginia, United States, November 10, 2011

Abstract

This document contains a tutorial on Copilot and its accompanying tools. We do not attempt to give a complete, formal description of Copilot (references are provided in the bibliography), rather we aim at demonstrating the fundamental concepts of the language by using idiomatic expositions and examples.

Contents

This research is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office.

1 Preliminaries

Copilot is embedded into the functional programming language Haskell Jones [2002]. A working knowledge of Haskell is necessary to use Copilot effectively; a variety of books and free web resources introduce Haskell. Copilot uses Haskell language extensions specific to the Glasgow Haskell Compiler (GHC); hence in order to start using Copilot, you must first install an up-to-date version of GHC. (The minimal required version is 7.0.) The easiest way to do this is to download and install the Haskell Platform, which is freely distributed from here:

`http://hackage.haskell.org/platform`

After having installed the Haskell Platform, Copilot is downloaded and installed by executing the following command:

```
> cabal install copilot
```

This should, if everything goes well, install Copilot on your system.

Copilot is distributed throughout a series of packages at Hackage:

- copilot-language: Contains the language front-end.
- copilot-core: Contains an intermediate representation for Copilot programs (shared by all back-ends).
- copilot-c99: A back-end for Copilot targeting C99 (based on Atom, `http://hackage.haskell.org/package/atom`).
- copilot-sbv: A back-end for Copilot targeting C99 (based on SBV, `http://hackage.haskell.org/package/sbv`).
- copilot-libraries: A set of utility functions for Copilot, including a clock-library, a linear temporal logic framework, a voting library, and a regular expression framework.
- copilot-cbmc: A driver for proving the correspondence between code generated by the copilot-c99 and copilot-sbv back-ends.

Many of the examples in this paper can be found at `https://github.com/leepike/Copilot/tree/copilot2.0/Examples`.

To use the language, your Haskell module should contain the following import:

```
import Language.Copilot
```

To use the back-ends, import, them, respectively:

```
import Copilot.Compile.C99
import Copilot.Compile.SBV
```

If you need to use functions defined in the Prelude that are redefined by Copilot (e.g., arithmetic operators), import the Prelude as qualified:

```
import qualified Prelude as P
```

2 Domain

Copilot is a domain-specific language tailored to programming *runtime monitors* for *hard real-time*, *distributed*, *reactive systems*. Briefly, a runtime monitor is program that runs concurrently with a target program with the sole purpose of assuring that the target program behaves in accordance with a pre-established specification. Copilot is a language for writing such specifications.

A reactive system is a system that responds continuously to its environment. All data to and from a reactive system is communicated progressively during execution. Reactive systems differ from transformational systems which transforms data in a single pass and then terminate, as for example compilers and numerical computation software.

A hard real-time system is a system that has a statically bounded execution time and memmory usage. Typically, hard real-time systems are used in mission-critical software, such as avionics, medical equipment, and nuclear power plants; hence, occasional dropouts in the response time or crashes are not tolerated.

A distributed system is a system which is layered out on multiple pieces of hardware. The distributed systems we consider are all synchronized, i.e., each component agree on a shared global clock.

3 Language

Copilot is a pure declarative language; i.e., expressions are free of side-effects and satisfies referential transparency. A program written in Copilot, which from now on will be referred to as a *specification*, has a cyclic behavior, where each cycle consists of a fixed series of steps:

- Sample external variables, arrays, and functions.

- Update internal variables.
- Fire external triggers. (In case the specification is violated.)

We refer to a single cycle as an *iteration*.

All transformation of data in Copilot is propagated through streams. A stream is an infinite, ordered sequence of values which must conform to the same type. E.g. we have the stream of Fibonacci numbers:

$$s_{fib} = \{0, 1, 1, 2, 3, 5, 8, 13, 21, \dots\}$$

We denote the nth value of the stream s as $s(n)$, and the first value in a sequence s as $s(0)$. For example, for s_{fib} we have that $s_{fib}(0) = 1$, $s_{fib}(1) = 1$, $s_{fib}(2) = 2$, and so forth.

Constants as well as arithmetic, boolean, and relational operators are lifted to work pointwise on streams:

```
x :: Stream Int32        y :: Stream Int32        z :: Stream Int32
x = 5 + 5                y = x * x                z = x == 10 && y < 200
```

Here the streams `x`, `y`, and `z` are simply *constant streams*:

$$\texttt{x} \rightsquigarrow \{10, 10, 10, \dots\},\ \texttt{y} \rightsquigarrow \{100, 100, 100, \dots\},\ \texttt{z} \rightsquigarrow \{\mathrm{T},\ \mathrm{T},\ \mathrm{T},\ \dots\}$$

Two types of *temporal* operators are provided, one for delaying streams and one for looking into the future of streams:

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

Here `xs ++ s` prepends the list `xs` at the front of the stream `s`. For example the stream `w` defined as follows, given our previous definition of `x`:

```
w = [5,6,7] ++ x
```

evaluates to the sequence $\texttt{w} \rightsquigarrow \{5, 6, 7, 10, 10, 10, \dots\}$. The expression `drop k s` skips the first `k` values of the stream `s`, returning the remainder of the stream. For example we can skip the first two values of `w`:

```
u = drop 2 w
```

which yields the sequence $\texttt{u} \rightsquigarrow \{7, 10, 10, 10, \dots\}$.

3.1 Streams as Lazy-lists

A key design choice in Copilot is that streams should mimic *lazy lists*. In Haskell, the lazy-list of natural numbers can be programmed like this:

```
nats_ll :: [Int32]
nats_ll = [0] ++ zipWith (+) (repeat 1) nats_ll
```

As both constants and arithmetic operators are lifted to work pointwise on streams in Copilot, there is no need for `zipWith` and `repeat` when specifying the stream of natural numbers:

```
nats :: Stream Int32
nats = [0] ++ (1 + nats)
```

In the same manner, the lazy-list of Fibonacci numbers can be specified as follows:

```
fib_ll :: [Int32]
fib_ll = [1, 1] ++ zipWith (+) fib_ll (drop 1 fib_ll)
```

In Copilot we simply throw away `zipWith`:

```
fib :: Stream Int32
fib = [1, 1] ++ (fib + drop 1 fib)
```

Copilot specifications must be *causal*, informally meaning that stream values cannot depend on future values. For example, the following stream definition is allowed:

```
h :: Stream Word64
h = drop 2 g
  where g = f
        f = [0,1,2] ++ f
```

But if instead `h` is defined as `h = drop 4 g`, then the definition is disallowed. While an analogous stream is definable in a lazy language, we bar it in Copilot, since it requires future values of `g` and `f` to be generated before producing values for `h`. This is not possible since Copilot programs may take inputs in real-time from the environment (see Section 3.5).

3.2 Functions on Streams

Given that constants and operators work pointwise on streams, we can use Haskell as a macro-language for defining functions on streams. The idea of using Haskell as a macro language is powerful, since Haskell is a general-purpose higher-order functional language.

x_i:	y_{i-1}:	y_i:
F	F	F
F	T	T
T	F	T
T	T	F

```
latch :: Stream Bool -> Stream Bool
latch x = y
  where
  y = if x then not z else z
    where
    z = [False] ++ y
```

Figure 1: A latch. The specification is provided at the left and the implementation is provided at the right.

Example 1:
We define the function, `even`*, which given a stream of integers returns a boolean stream which is true whenever the input stream contains an even number, as follows:*

```
even :: Stream Int32 -> Stream Bool
even x = x `mod` 2 == 0
```

Applying `even` *on* `nats` *(defined above) yields the sequence* $\{T, F, T, F, T, F, \dots\}$.

If a function is required to return multiple results, we simply use plain Haskell tuples:

Example 2:
We define complex multiplication as follows:

```
mul_comp
  :: (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
(a, b) `mul_comp` (c, d) = (a * c - b * d, a * d + b * c)
```

Here `a` *and* `b` *represent the real and imaginary part of the left operand, and* `c` *and* `d` *represent the real and imaginary part of the right operand.*

3.3 Stateful Functions

In addition to pure functions, such as `even` and `mul_comp`, Copilot also facilitates *stateful* functions. A *stateful* function is function which has an internal state, e.g. as a latch (as in electronic circuits) or a low/high-pass filter (as in a DSP).

Example 3:
We consider a simple latch, as described in Farhat [2004], with a single input and a boolean state. Whenever the input is true the internal state is reversed.

$\texttt{inc}_i$:	$\texttt{reset}_i$:	$\texttt{cnt}_i$:
F	F	$\texttt{cnt}_{i-1}$
F	T	0
T	F	$\texttt{cnt}_{i-1} + 1$
T	T	0

```
counter :: Stream Bool -> Stream Bool
        -> Stream Int
counter inc reset = cnt
  where
    cnt | reset     = 0
        | inc       = z + 1
        | otherwise = z
    z = [0] ++ cnt
```

Figure 2: A resettable counter. The specification is provided at the left and the implementation is provided at the right.

The operational behavior and the implementation of the latch is shown in Figure 1.[1]

Example 4:
We consider a resettable counter with two inputs, `inc` *and* `reset`*. The input* `inc` *increments the counter and the input* `reset` *resets the counter. The internal state of the counter,* `cnt`*, represents the value of the counter and is initially set to zero. At each cycle, i, the value of* $\texttt{cnt}_i$ *is determined as shown in the left table in Figure 2.*

3.4 Types

Copilot is a typed language, where types are enforced by the Haskell type system to ensure generated C programs are well-typed. Copilot is *strongly typed* (i.e., type-incorrect function application is not possible) and *statically typed* (i.e., type-checking is done at compile-time). The base types are Booleans, unsigned and signed words of width 8, 16, 32, and 64, floats, and doubles.[2] All elements of a stream must belong to the same base type. These types have instances for the class `Typed a`, used to constrain Copilot programs.

We provide a `cast` operator

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

that casts from one type to another. The cast operator is only defined for casts that do not lose information, so an unsigned word type `a` can only be cast to another unsigned type at least as large as `a` or to a signed word type strictly larger than `a`. Signed types cannot be cast to unsigned types but can be cast to signed types at least as large.

[1] In order to use conditionals (if-then-else's) in Copilot specifications, as in Figure 1, or guards, as in Figure 2, the GHC language extension `RebindableSyntax` must be set on.

[2] Floats and doubles are supported by the Atom back-end but not the SBV back-end.

3.5 Interacting With the Target Program

All interaction with the outside world is done by sampling *external variables* and by evoking *triggers*. External variables are variables that are defined outside Copilot and which reflect the visible state of the target program that we are monitoring. Analogously, triggers are functions that are defined outside Copilot and which are evoked when Copilot needs to report that the target program has violated a specification constraint.

External variables are defined by using the `extern` construct:

```
extern :: Typed a => String -> Stream a
```

It takes the name of an external variable and generates a stream by sampling the variable at each clock cycle. For example,

```
sumExterns :: Stream Word64
sumExterns = let ex1 = extern "e1"
                 ex2 = extern "e2"
             in  ex1 + ex2
```

is a stream that takes two external variables `e1` and `e2` and adds them. Sometimes, type inference cannot infer the type of an external variable. For example, in the stream definition

```
extEven :: Stream Bool
extEven = extern "x" `mod` 2 == 0
```

the type of `extern "x"` is ambiguous. For convenience, typed `extern` functions are provided, e.g., `externW8` or `externI64` denoting an external unsigned 8-bit word or signed 64-bit word, respectively.

Triggers are defined by using the `trigger construct`:

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

The first parameter is the name of the external function, the second parameter is the guard which determines when the trigger should be evoked, and the third parameter is a list of arguments which is passed to the trigger when evoked. Triggers can be combined into a specification by using the *do*-notation:

```
spec :: Spec
spec = do
  trigger "f" (even nats) [arg fib, arg (nats * nats)]
  trigger "g" (fib > 10) []
  let x = externW32 "x"
  trigger "h" (x < 10) [arg x]
```

The order in which the triggers are defined is irrelevant.

Example 5:
We consider an engine controller with the following property: If the temperature rises more than 2.3 degrees within 0.2 seconds, then the engine should be shut off immediately. Assuming that the global sample rate is 0.2 seconds, we can define a monitor that surveys the above property:

```
propTempRiseShutOff :: Spec
propTempRiseShutOff = trigger "over_temp_rise" (overTempRise ==> not running) []
  where
  temps        = [0, 0, 0] ++ (extern "temp" :: Stream Float)
  overTempRise = drop 2 temps > const 2.3 + temps
  running      = extern "running"
```

Here, we assume that the external variable `temp` *denotes the temperature of the engine and the external variable* `running` *indicates whether the engine is running. The external function* `over_temp_rise` *is called without any arguments if the temperature rises more than 2.3 degrees within 0.2 seconds and the engine is not shut off.*

Besides variables, external arrays and arbitrary functions can be sampled. The external array construct has the type

```
externArray :: (Typed a, Typed b, Integral a)
            => String -> Stream a -> Stream b
```

The construct takes the name of an array and a stream that generates indexes for the array (of integral type). For example,

```
extArr :: Stream Word32
extArr = externArray "arr1" arrIdx
  where
  arrIdx :: Stream Word8
  arrIdx = [0] ++ (arrIdx + 1) `mod` 4
```

`extArr` is a stream of values drawn from an external array containing 32-bit unsigned words. The array is indexed by an 8-bit variable. The index is ensured to be less than four by using modulo arithmetic.

Example 6:
Say we have defined a lookup-table (in C99) of a discretized continuous function that we want to use within Copilot:

```
double someTable[] = { 3.5, 3.7, 4.5, ... };
```

We can use the table in a Copilot specification as follows:

```
lookupSomeTable :: Stream Int -> Stream Stream Double
lookupSomeTable k = externArray "someTable" k
```

Given the following values for `k`, $\{1, 0, 2, 2, 1, \dots\}$, *the output of* `lookupSomeTable k` *would be* $\{3.7, 3.5, 4.5, 4.5, 3.7, \dots\}$.

The constructor `externFun` also takes two arguments: the name of the external function and a list of arguments that are provided to the function.

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

Each argument to an external function is given by a Copilot stream. For example,

```
func :: Stream Word16
func = externFun "f" [funArg $ externW8 "x", funArg nats]
  where
  nats :: Stream Word16
  nats = [0] ++ nats + 1
```

samples a function in C that has the prototype

```
uint16_t f(uint8_t x, uint16_t nats);
```

Both external arrays and functions must, like external variables, be defined in the target program that is monitored. Additionally external functions must be without side effects, as the monitor can cause undesired side-effects when calling the function.

3.6 Explicit Sharing

```
s1 = let x = nats + nats                s2 = local (nats * nats) $
     in x * x                                 \ x -> x + x
```

Figure 3: Implicit sharing (to the left) versus explicit sharing (to the right).

Copilot facilitates sharing in expressions by the *local*-construct:

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

The local construct works similar to *let*-bindings in ordinary Haskell. From a semantic point of view, the streams `s1` and `s2` from Figure 3 are identical. As we will see in Section 5, however, certain advanced Copilot programs may force the compiler to build syntax trees that blow up exponentially. In such cases, using explicit sharing helps to avoid this problem.

4 Tools

Copilot comes with a variety of tools, including a pretty-printer, an interpreter, a two compilers targeting C, and a verifier front-end. In the following section, we will demonstrate some of these tools and their usage.

4.1 Pretty-Printing

Pretty-printing is straightforward. For some specification `spec`,

```
prettyPrint spec
```

returns the specification after static macro expansion. Pretty-printing can provide some indication about the complexity of the specification to be evaluated. Specifications that are built by recursive Haskell programs (e.g., the majority voting example in Section 5) can generate expressions that are quite large. Very large expressions can take significant time to interpret or compile.

4.2 Interpreting Copilot

Suppose we want to interpret the engine controller from Example 5. Before we can do so, we must define the external variables `temp` and `running` as infinite Haskell lists:

```
temp :: [Float]
temp = map exp [0.2, 0.4 ..]

running :: [Bool]
running = repeat True
```

We now evoke the interpreter as follows (e.g. in GHCI, the GHC compiler's interpreter for Haskell):

```
GHCI> interpret 10
                [var "temp" temp, var "running" running]
                propTempRiseShutOff
```

The first argument to the function *interpret* is the number of iterations that we want to evaluate. The second argument is a list of inputs. Each input takes a name and an infinite list of values (which must conform to the type of the input). The third argument is the specification that we wish to interpret.

If a Copilot program contains no external variables, then `interpret` is given the empty list `[]`.

The interpreter outputs the values of the arguments passed to the trigger, if its guard is true, and `--` otherwise. For example, consider the following Copilot program:

```
spec = do
  trigger "trigger1" (even nats) [arg nats, arg $ odd nats]
  trigger "trigger2" (odd nats) [arg nats]
```

where `nats` is the stream of natural numbers, and `even` and `odd` are functions that take a stream and return whether the point-wise values are even or odd, respectively. The output of

```
interpret 10 [] spec
```

is as follows:

```
trigger:   trigger2:
(0,false)  --
--         (1)
(2,false)  --
--         (3)
(4,false)  --
--         (5)
(6,false)  --
--         (7)
(8,false)  --
--         (9)
```

In programs containing external arrays, an infinite *list of lists* is given, simulating how the array changes in each tick. For example, for a program containing an array `arr` of unsigned 32-bit integers, of length four, we can define `arrInterp`

```
arrInterp :: [Word32]
arrInterp =
  let shiftl ls = ls : shiftl (tail ls ++ [head ls]) in
  shiftl [0..3]
```

to generate the list

```
[[0,1,2,3],[1,2,3,0],[2,3,0,1],[3,0,1,2] ... ]
```

We can then use `arrInterp` as an input to simulate the program containing it.

```
interpret 10 [ array "arr" arrInterp ] spec
```

Note that the index into the array must not be greater than three!

External functions can be modeled by a stream that simulates the behavior of the function. for example, for an external function that takes an external variable and a Copilot stream as arguments

```
func :: Stream Word16
func = externFun "f" [funArg $ externW8 "x", funArg nats]
```

we can define a stream that also takes an external variable `x` that is an 8-bit unsigned word and the stream `nats`. Suppose we decide that our simulation stream adds its two arguments. To interpret the function, we call

```
x :: Word8
x = [0..]

interpret 10 [ func "func0" (cast (externW8 "x") + nats)
             , var "x" x ]
          spec
```

Sometimes it is convenient to observe the behavior of a stream without defining a trigger. We can do so by using an observer. For example:

```
spec :: Spec
spec = observer ‘‘obs’’ nats
```

can be interpreted using

```
interpret 5 [] spec
```

as usual. Observers can be combined in larger Copilot programs. For example, consider the following:

```
spec :: Spec
spec = do
  let x = externW8 "x"
  trigger "trigger" true [arg $ x < 3]
  observer "debug_x" x
```

Interpreting `spec` as follows

```
interpret 10 [var "x" [0 :: Word8 ..]] foo
```

yields

```
trigger:  debug_x:
(true)    0
(true)    1
(true)    2
(false)   3
(false)   4
(false)   5
(false)   6
(false)   7
(false)   8
(false)   9
```

4.3 Compiling Copilot

Compiling the engine controller from Example 5 is straightforward. First, we pick a back-end to compile to (see Section 1 for a brief list of back-ends; we compile to Atom below), import it, and compile as follows:[3]

```
reify spec >>= compile defaultParams
```

(The compile function takes a parameter to rename the generated C files; `defaultParams` is the default, in which there is no renaming.)

The compiler now generates two files:

- "copilot.c" —
- "copilot.h" —

The file named "copilot.h" contains prototypes for all external variables, functions, and arrays, and contains a prototype for the "step"-functions which evaluates a single iteration.

```
/* Generated by Copilot Core v. 0.1 */

#include <stdint.h>
#include <stdbool.h>

/* Triggers (must be defined by user): */

void over_temp_rise();

/* External variables (must be defined by user): */
```

[3]Two explanations are in order: (1) `reify` is an optimization that improves sharing in the expressions to be compiled, improving efficiency Gill [2009], and `>>=` is a higher-order operator that takes the result of reification and "feeds" it to the compile function.

```
extern float temp;
extern bool running;

/* Step function: */

void step();
```

Using the prototypes in "copilot.h" we can build a driver as follows:

```
/* driver.c */
#include <stdio.h>
#include "copilot.h"

bool running = true;
float temp = 1.1;

void over_temp_rise()
{
  printf("The trigger has been evoked!\n");
}

int main (int argc, char const *argv[])
{
  int i;

  for (i = 0; i < 10; i++)
  {
    printf("iteration: %d\n", i);
    temp = temp * 1.3;
    step();
  }

  return 0;
}
```

Running "gcc copilot.c driver.c -o prop" gives a program "prop", which when executed yields the following output:

```
iteration: 0
iteration: 1
iteration: 2
iteration: 3
iteration: 4
iteration: 5
iteration: 6
iteration: 7
```

```
The trigger has been evoked!
iteration: 8
The trigger has been evoked!
iteration: 9
The trigger has been evoked!
```

4.4 QuickCheck

QuickCheck Claessen and Hughes [2000] is a library originally developed for Haskell such that given a property, it generates random inputs to test the property. We provide a similar tool for checking Copilot specifications. Currently, the tool is implemented to check the copilot-c99 back-end against the interpreter. The tool generates a random Copilot specification, and for some user-defined number of iterations, the output of the interpreter is compared against the output of the compiled C program. The user can specify weights to influence the probability at which expressions are generated.

If you have installed Copilot, you can execute the quickCheck tests by executing the program `CopilotC99Test`. The default installation for the executable is in `$HOME/.cabal/bin`. Assuming the executable is in your path, simply execute it. It will direct the user to enter the number of specifications to test. The program will then generate that many random specifications, testing the output of the interpreter against the executed C program. By default, it tests the outputs for ten iterations.

4.5 Verification

"Who watches the watchmen?" Nobody. For this reason, monitors in ultra-critical systems are the last line of defense and cannot fail. Here, we outline our approach to generate high-assurance monitors. First, as mentioned, the compiler is statically and strongly typed, and by implementing an eDSL, much of the infrastructure of a well-tested Haskell implementation is reused. We have described our custom QuickCheck engine. We have tested millions of randomly-generated programs between the compiler and interpreter with this approach.

Additionally, Copilot includes a tool to generate a driver to prove the equivalence between the copilot-c99 and copilot-sbv back-ends that each generate C code (similar drivers are planned for future back-ends). To use the driver, first import the following module:

```
import qualified Copilot.Tools.CBMC as C
```

(We import it using the `qualified` keyword to ensure no name space collisions.) Then in GHCI, just like with compilation, we execute

```
reify spec >>= C.genCBMC C.defaultParams
```

This generates two sets of C sources, one compiled through the copilot-c99 back-end and one through the copilot-sbv back-end. In addition, a driver (that is, a `main` function) is generated that executes the code from each back-end. The driver has the following form:

```
int main (int argc, char const *argv[])
{
  int i;

  for (i = 0; i < 10; i++)
  {
    sampleExterns();
    atm_step();
    sbv_step();
    assert(atm_i == sbv_i);
  }

  return 0;
}
```

This driver executes the two generated programs for ten iterations, which is the default value. That default can be changed; for example:

```
reify spec >>=
  C.genCBMC C.defaultParams {C.numIterations = 20}
```

The above executes the generated programs for 20 executions.

The verification depends on an open-source model-checker for C source-code originally developed at Carnegie Mellon University Clarke et al. [2004]. A license for the tool is available. [4] CBMC must be downloaded and installed separately; CBMC is actively maintained at the time of writing, and is available for Windows, Linux, and Mac OS.

CBMC symbolically executes a program. With different options, CBMC can be used to check for arithmetic overflow, buffer overflow/underflow, floating-point NaN results, and division by zero. Additionally, CBMC can attempt to verify arbitrary `assert()` statements placed in the code. In our case, we wish

[4] `http://www.cprover.org/cbmc/LICENSE`. It is the user's responsibility to ensure their use conforms to the license.

```
majorityPure :: Eq a => [a] -> a
majorityPure []     = error "majorityPure: empty list!"
majorityPure (x:xs) = majorityPure' xs x 1

majorityPure' []     can _   = can
majorityPure' (x:xs) can cnt =
  let
    can' = if cnt == 0 then x else can
    cnt' = if cnt == 0 || x == can then succ cnt else pred cnt
  in
    majorityPure' xs can' cnt'
```

Figure 4: The first pass of the majority vote algorithm in Haskell.

```
aMajorityPure :: Eq a => [a] -> a -> Bool
aMajorityPure xs can = aMajorityPure' 0 xs can > length xs `div` 2

aMajorityPure' cnt []     _   = cnt
aMajorityPure' cnt (x:xs) can =
  let
    cnt' = if x == can then cnt+1 else cnt
  in
    aMajorityPure' cnt' xs can
```

Figure 5: The second pass of the majority vote algorithm in Haskell.

to verify that on each iteration, for the same input variables, the two back-ends have the same state.

CBMC proves that for all possible inputs, the two programs have the same outputs for the number of iterations specified. The time-complexity of CBMC is exponential with respect to the number of iterations. Furthermore, CBMC cannot guarantee equivalence beyond the fixed number of iterations.

After generating the two sets of C source files, CBMC can be executed on the file containing the driver; for example,

```
cbmc cbmc_driver.c
```

5 Extended Example: The Boyer-Moore Majority-Vote Algorithm

In this section we demonstrate how to use Haskell as an advanced macro language on top of Copilot by implementing an algorithm for solving the voting problem in Copilot.

Reliability in mission critical software is often improved by replicating the same computations on separate hardware and by doing a vote in the end based on the output of each system. The majority vote problem consists of determining

if in a given list of votes there is a candidate that has more than half of the votes, and if so, of finding this candidate.

The Boyer-Moore Majority Vote Algorithm Moore and Boyer [1981], Hesselink [2005] solves the problem in linear time and constant memory. It does so in two passes: The first pass eliminates every, but a single, candidate; and the second pass asserts that the found candidate indeed holds a majority.

Without going into details of the algorithm (references are provided in the bibliography), the first pass can be implemented in Haskell as shown in Figure 4. The second pass, which simply checks that a candidate has more than half of the votes, is straightforward to implement and is shown in Figure 5. E.g. applying `majorityPure` on the string `AAACCBBCCCBCC` yields `C`, which `aMajorityPure` can confirm is in fact a majority.

```
majority :: (Eq a, Typed a) => [Stream a] -> Stream a
majority []     = error "majority: empty list!"
majority (x:xs) = majority' xs x 1

majority' []     can _   = can
majority' (x:xs) can cnt =
  local
    (if cnt == 0 then x else can) $
      \ can' ->
        local (if cnt == 0 || x == can then cnt+1 else cnt-1) $
          \ cnt' ->
            majority' xs can' cnt'
```

Figure 6: The first pass of the majority vote algorithm in Copilot.

```
aMajority :: (Eq a, Typed a) => [Stream a] -> Stream a -> Stream Bool
aMajority xs can = aMajority' 0 xs can > (fromIntegral (length xs) `div` 2)

aMajority' cnt []     _   = cnt
aMajority' cnt (x:xs) can =
  local
    (if x == can then cnt+1 else cnt) $
      \ cnt' ->
        aMajority' cnt' xs can
```

Figure 7: The second pass of the majority vote algorithm in Copilot.

When implementing the majority vote algorithm for Copilot, we can use reuse almost all of the code from the Haskell implementation. However, as functions in Copilot in reality are macros that are expanded at compile time, care must be taken in order to avoid an explosion in the code size. Hence, instead of using Haskell's built-in *let*-blocks, we use explicit sharing, as described

in Section 3.6. The Copilot implementations of the first and the second pass are given in Figure 6 and Figure 7 respectively. Comparing the Haskell implementation with the Copilot implementation, we see that the code is almost identical, except for the type signatures and the explicit sharing annotations.

Further Reading

For detailed background information on Copilot we refer to Pike et al. [2010] and Pike et al. [2011].

Acknowledgement

The authors are grateful for NASA Contract NNL08AD13T to Galois Inc. and the National Institute of Aerospace, which partially supported this work.

References

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 168–176. Springer, 2004.

H.A. Farhat. *Digital design and computer organization*. Number v. 1 in Digital Design and Computer Organization. CRC Press, 2004. ISBN 9780849311918. URL `http://books.google.com/books?id=QypINJ4oRI8C`.

Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, September 2009.

Wim H. Hesselink. The boyer-moore majority vote algorithm, 2005.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

Strother J. Moore and Robert S. Boyer. MJRTY - A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, February 1981.

Lee Pike, Sebastian Niller, Alwyn Goodloe, and Nis Wegmann. Copilot: A hard real-time runtime monitor. To appear in the proceedings of the 1st Intl. Conference on Runtime Verification (RV'2010), 2010. Springer, 2010.

Lee Pike, Sebastian Niller, Alwyn Goodloe, and Nis Wegmann. Runtime verification for ultra-reliable systems. To appear at the 2nd International Conference on Runtime Verification, 2011.