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

An Introduction to Copilot

A tutorial to Copilot 3.0

Frank Dedden
dev@dedden.net

Alwyn Goodloe
a.goodloe@nasa.gov

Ivan Perez
ivan.perez@nianet.org

Macallan Cruff
mcruff@andrew.cmu.edu

Nis N. Wegmann
niswegmann@gmail.com

Lee Pike
leepike@galois.com

Chris Hathhorn
hathhorn@gmail.com

Sebastian Niller
sebastian.niller@gmail.com

Lauren Pick
lmpick@zoho.com

Georges-Axel Jaloyan
georges-axel.jaloyan@ens.fr

Hampton, Virginia, United States, April 9, 2019

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, 80LARC17C0004 and NNL09AA00A.

Acknowledgement

The authors are grateful for NASA Contract NNL08AD13T to Galois Inc. and the National Institute of Aerospace, which partially supported this work. Thanks to Lars Kuhtz, Benedetto Di Vito, Robin Morisset, Kaveh Darafsheh.

1 Introduction

Neither formal verification nor testing can ensure system reliability. Over 25 years ago, Butler and Finelli showed that testing alone cannot verify the reliability of ultra-critical software [BF93]. *Runtime verification* (RV) [GP10], where monitors detect and respond to property violations at runtime, has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. Technically speaking, a RV monitor takes a logical specification ϕ and execution trace τ of state information of the system under observation (SUO) and decides whether τ satisfies ϕ. The *Simplex Architecture* [Sha01] provides a model architectural pattern for RV, where a monitor checks that the executing SUO satisfies a specification and, if the property is violated, the RV system will switch control to a more conservative component that can be assured using conventional means that *steers* the system into a safe state. *High-assurance* RV provides an assured level of safety even when the SUO itself cannot be verified by conventional means.

Copilot is a RV framework for specifying and generating monitors for C programs that is embedded into the functional programming language Haskell [Jon02]. 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).

1.1 Target Application 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 are communicated progressively during execution. Reactive systems differ from transformational systems which transform 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 memory 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., all components agree on a shared global clock.

1.2 Installation

Before downloading the copilot source code, you must first install an up-to-date version of GHC (the minimal required version is 8.0). 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`

Because Copilot compiles to C code, you must also install a C compiler such as GCC (`https://gcc.gnu.org/install/`). After having installed the Haskell Platform and C compiler, Copilot can be downloaded and installed in the following two ways:

- **Using Cabal:** Copilot can be downloaded and installed by executing the following command:

  ```
  > cabal install copilot
  ```

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

- **Using Git:**

  ```
  git clone https://github.com/Copilot-Language/Copilot.git
  git submodule update --init --remote
  make
  ```

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

```
import Language.Copilot
```

To use the C99 back-end, import it:

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

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

```
import qualified Prelude as P
```

1.3 Structure

Copilot is distributed through a series of packages at Hackage:

- copilot-language: Contains the language front-end.
- copilot-theorem: Contains extensions to the language for proving properties about Copilot programs using various SMT solvers and model checkers.
- copilot-core: Contains an intermediate representation for Copilot programs.
- copilot-c99: A back-end for Copilot targeting C99.
- 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.

Many of the examples in this paper can be found at `https://github.com/Copilot-Language/Copilot/tree/master/Examples`.

1.4 Sampling

The idea of sampling representative data from a large set of data is well established in data science and engineering. For instance, in digital signal processing, a signal such as music is sampled at a high enough rate to obtain enough discrete points to represent the physical sound wave. The fidelity of the recording is dependant on the sampling rate. Sampling a state variable of an executing program is similar, but variables are rarely continuous signals so they lack the nice properties of continuity. Monitoring based on sampling state-variables has historically been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed software, monitoring via sampling may lead to false positives and false negatives [DDE08]. For example, consider the property $(0; 1; 1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take. If the monitor samples the variable at the inappropriate time, then both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is $0, 1, 1, 0, 1, 1$, then an observation of $0, 1, 1, 1, 1$ is a false negative by skipping a value, and if the actual sequence is $0, 1, 0, 1, 1$, then an observation of $0, 1, 1, 0, 1, 1$ is a false positive by sampling a value twice.

However, in a hard real-time context, sampling is a suitable strategy. Often, the purpose of real-time programs is to deliver output signals at a predicable rate and properties of interest are generally data-flow oriented. In this context, and under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. A false positive is possible, for example, if the program does not execute according to its schedule but just happens to have the expected values when sampled. If

a monitor samples an unacceptable sequence of values, then either the program is in error, the monitor is in error, or they are not synchronized, all of which are faults to be reported.

Most of the popular runtime monitoring frameworks inline monitors in the observed program to avoid the aforementioned problems with sampling. However, inlining monitors changes the real-time behavior of the observed program, perhaps in unpredicable ways. Solutions that introduce such unpredictability are not a viable solution for ultra-critical hard real-time systems. In a sampling-based approach, the monitor can be integrated as a separate scheduled process during available time slices (this is made possible by generating efficient constant-time monitors). Indeed, sampling-based monitors may even be scheduled on a separate processor (albeit doing so requires additional synchronization mechanisms), ensuring time and space partitioning from the observed programs. Such an architecture may even be necessary if the monitored program is physically distributed.

When deciding whether to use Copilot to monitor systems that are not hard real-time, the user must determine if sampling is suitable to capture the errors and faults of interest in the SUO. In many cyber-physical systems, the trace being monitored is coming from sensors measuring physical data such as GPS coordinates, air speed, and actuation signals. These continuous signals do not change abruptly so as long as it is sampled at a suitable rate, that usually must be determined experimentally, sampling is sufficient.

2 Interpreting and Compiling

The Copilot RV framework comes with both an interpreter and a compiler.

2.1 Interpreting Copilot

Assume we are currently in a directory containing a `.hs` file with our specification (`Spec.hs` in this case), and that Copilot is installed globally. If we want to interpret the specification, we need to start the GHC Interpreter with the file as an argument:

```
$ ghci Spec.hs
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/user/.ghc/ghci.conf
[1 of 1] Compiling Spec             ( Spec.hs, interpreted )
Ok, one module loaded.
*Spec >
```

This launches `ghci`, the Haskell interpreter, with `Spec.hs` loaded. It provides us with a prompt, recognisable by the `>` sign. Lets assume that our file contains one specification, called `spec`. We can interpret this using the `interpret`-function:

```
*Spec > interpret 10 spec
```

The first argument to the function *interpret* is the number of iterations that we want to evaluate. The second argument is the specification (of type `Spec`) that we wish to interpret.

The interpreter outputs the values of the arguments passed to the trigger, if its guard is true, and `--` otherwise. We will discuss triggers in more detail later, but for now, just know that they produce an output only when the guard function is true. 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 the guard functions that take a stream and return whether the point-wise values are even or odd, respectively. The lists at the end of the trigger represent the values the trigger will output when the guard is true. The output of

```
interpret 10 spec
```

is as follows:

```
trigger1:   trigger2:
(0,false)  --
--         (1)
```

```
(2,false)  --
--         (3)
(4,false)  --
--         (5)
(6,false)  --
--         (7)
(8,false)  --
--         (9)
```

Note that trigger1 outputs both the number and whether that number is odd, while trigger2 only outputs the number. This output reflects the arguments passed to them.

2.2 Compiling Copilot

Compiling a Copilot specification is straightforward. Currently Copilot supports one back-end, `copilot-c99` that creates constant-space C99 code. Using the back-end is rather easy, as it just requires one to import it in their Copilot specification file:

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

Importing the back-end provides us with the `compile`-function, which takes a prefix as its first parameter and a *reified* specification as its second. When inside `ghci`, with our file loaded, we can generate output code by executing: [1]

```
reify spec >>= compile "monitor"
```

This generates two output files:

- `<prefix>.c`: C99 file containing the generated code and the `step()` function. This should be compiled by the C compiler, and included in the final binary.
- `<prefix>.h`: Header providing the public interface to the monitor. This file should be included from your main project.

Please refer to the complete example 4 for more on detail to use the monitor in your C program.

3 Language

Copilot is embedded into the functional programming language Haskell [Jon02], and a working knowledge of Haskell is necessary to use Copilot effectively. Copilot is a pure declarative language;

[1] Two explanations are in order: (1) `reify` allows sharing in the expressions to be compiled [Gil09], and `>>=` is a higher-order operator that takes the result of reification and "feeds" it to the compile function.

i.e., expressions are free of side-effects and are referentially transparent. 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 and arrays.
- Update internal variables.
- Fire external triggers. (In case the specification is violated.)
- Update observers (for debugging purpose).

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

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 $s_{fib}(0) = 0$, $s_{fib}(1) = 1$, $s_{fib}(2) = 1$, and so forth.

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

```
x :: Stream Int32
x = 5 + 5

y :: Stream Int32
y = x * x

z :: Stream Bool
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 `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 `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 in Haskell 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:

```
f :: Stream Word64
f = [0,1,2] ++ f

g :: Stream Word64
g = drop 2 f
```

But if instead `g` is defined as `g = drop 4 f`, 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 `f` to be generated before producing values for `g`. This is not possible since Copilot programs may take inputs in real-time from the environment (see Section 3.6).

Structs require some special attentation in Copilot, as we cannot magically import the definition of the struct in Copilot. In this section we discuss the steps that need to be taken by following the code of `Struct.hs` in the `Examples` directory of the Copilot distribution, or the repository [2].

Let's assume that we have defined a 2d-vector type in our C code:

```
struct vec {
  float x;
  float y;
};
```

For us to be able to use this vector inside Copilot, we need to follow a number of steps:

1. Enable `DataKinds` compiler extension.
2. Define a datatype to mimic the C definition.
3. Write an instance of the `Struct` class, containing a definition for the struct name and function to translate the fields to a heterogeneous list.
4. Write an instance of the `Typed` class.

Enabling compiler extensions

First and foremost, we need to enable the `DataKinds` extension to GHC, by putting:

```
{-# LANGUAGE DataKinds #-}
```

at the top of our specification file. This allows us to define *kinds*, which are the types of types. Our datatype needs to carry the names of the fields in C as well. Using the `DataKinds` extension we are able to write the names of the fields as part of our types.

Defining the datatype

A suitable representation of structs in Haskell is provided by the *record-syntax*, this allows us to use named fields as part of the datatype. For Copilot this is not enough though: we still need to define the names of the fields in our C code. Therefore we introduce new `Field` datatype, which takes two arguments: the name of field, and it's type. Now we can mimic our vector struct in Copilot as follows:

```
data Vec = Vec
  { x :: Field "x" Float
  , y :: Field "y" Float
  }
```

[2] `https://github.com/Copilot-Language/Copilot/blob/master/Examples/Struct.hs`

Here we created two fields, x and y, each with their corresponding C names and types. Note that the name inside Haskell and the C names do not necessarily need to match, nor is it always possible to have them match. For type-safety, inside Copilot we will typically only use the Haskell level names (i.e. the unquoted ones). The C names are only used by Copilot internally.

Instance of `Struct`

Our next task is to inform Copilot about our new type, therefore we need to write and instance of the `Struct`-class. This class has the purpose of defining the datatype as a struct, it provides the code generator of Copilot the name of struct in C, and provides a function to translate the struct to a list of values:

```
instance Struct Vec where
  -- typename :: Vec -> String
  typename _ = "vec"  -- Name of the type in C

  -- Function to translate Vec to list of Value's, order should match struct.
  -- tovalues :: Vec -> [Value Vec]
  toValues v = [ Value Float (x v)
               , Value Float (y v)
               ]
```

Both definitions should be pretty self-explanatory. Note however that `Value a` is a wrapper around the `Field` datatype to hide the actual type of `Field`. It takes the type of the field, and the field itself as its arguments. The elements in the list should be in the same order as the fields in the struct.

Both `typename` and `toValues` have to be defined by the user, but neither should ever be used by the user. Both functions are only used by the code generator of Copilot.

Instance of `Typed`

In Copilot, streams can only of types that are instances of the `Typed` class. To be able to create streams of vectors, `Vec` needs to be an instance of `Typed` as well. The class only provides a `typeOf` function, returning the type:

```
instance Typed Vec where
  typeOf = Struct (Vec (Field 0) (Field 0))
```

For `Vec` this means we need to return something of the `Vec` type wrapped in the `Struct` constructor. In this case it does not matter what the values of the fields are, we just need to return something of the correct type.

Simple operations

Building streams of structs works like building any other stream, but we need to wrap the values of a struct using the `Field` constructor. The reason for this is quite straightforward: the fields of our struct are defined in terms of `Field`:

```
v :: Stream Vec
v = [ Vec (Field 0) (Field 1) ] ++ v
```

We can also turn a field of a struct into its own stream using the `(#)`-operator:

```
vx :: Stream Float
vx = v # x
```

Note the we use the Haskell level accessor `x` to retrieve the field from the stream of vectors.

Example code

Example 1:
Now that we defined all there is, we can make streams of structs. The following code has been taken from the `Struct.hs` *example, and shows the basic usage of structs.*

```
{-# LANGUAGE DataKinds #-}

module Struct where

import Language.Copilot
import Copilot.Compile.C99

import Prelude hiding ((>), (<), div, (++))


data Vec = Vec
  { x :: Field "x" Float
  , y :: Field "y" Float
  }

instance Struct Vec where
  typename _ = "vec"  -- Name of the type in C

  -- Function to translate Vec to list of Value's, order should match struct.
  toValues v = [ Value Float (x v)
               , Value Float (y v)
               ]

-- We need to provide an instance to Typed with a bogus Vec
```

```
instance Typed Vec where
  typeOf = Struct (Vec (Field 0) (Field 0))


vecs :: Stream Vec
vecs = [ Vec (Field 1) (Field 2)
       , Vec (Field 12) (Field 8)
       ] ++ vecs


spec = do
  -- Trigger that always executes, splits the vec into seperate args.
  trigger "split" true [arg $ vecs # x, arg $ vecs # y]
```

3.3 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.

Example 2:
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 3:
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.*

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
  z = [False] ++ y
```

	0	1	2	3	4
x	F	T	T	F	F
y	F	T	F	F	F

Figure 1: A latch [Example 3]. The specification function is provided at the left and the implementation in copilot is provided in the middle. The right shows an example of the latch, where x is $\{F, T, T, F, F, \dots\}$ and the initial value of y (used with x_0 to find y_0 since there is no y_{-1}) is False.

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

```
counter :: Stream Bool -> Stream Bool
        -> Stream Int32
counter inc reset = cnt
  where
  cnt = if reset then 0
          else if inc then z + 1
                 else z
  z = [0] ++ cnt
```

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

3.4 Stateful Functions

In addition to pure functions, such as `even` and `mul_comp`, Copilot also facilitates *stateful* functions. A *stateful* function is a 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 4:

We consider a simple latch, as described in [Far04], with a single input and a boolean state. A latch is a way of simulating memory in circuits by feeding back output gates as inputs. Whenever the input is true the internal state is reversed. The operational behavior and the implementation of the latch is shown in Figure 1.[3]

Example 5:

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] In order to use conditionals (i.e., if-then-else) in Copilot specifications, as in Figures 1 and 2, the GHC language extension `RebindableSyntax` must be set on.

3.5 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. 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.

There also exists an `unsafeCast` operator which allows casting from any type to any other (except from floating point numbers to integer types):

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

3.6 Interacting With the Target Program

All interaction with the outside world is done by sampling *external symbols* and by evoking *triggers*. External symbols are symbols that are defined outside Copilot and which reflect the visible state of the target program that we are monitoring. They include variables and arrays. 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. As discussed in Section 1.4, *sampling* is an approach for monitoring the state of an executing system based on sampling state-variables, while assuming synchrony between the monitor and the observed software. Copilot targets hard real-time embedded C programs so the state variables that are observed by the monitors are variables of C programs. Copilot monitors run either in the same thread or a separate thread as the system under observation and the only variables that can be observed are those that are made available through shared memory. This means local variables cannot be observed. Currently, Copilot supports basic C datatypes, arrays and structs. Combinations of each of those work as well: nested arrays, arrays of structs, structs containg arrays etc. All of these variables containing actual data; pointers to data are not supported by design.

Copilot has both an interpreter and a compiler.The compiler must be used to monitor an executing program. The Copilot reification process generates a C monitor from a Copilot specification. The variables that are observed in the C code must be declared as *external* variables in the monitor. The external variables have the same name as the variables being monitored in the C code are treated as shared memory. The interpreter is intended for exploring ideas and algorithms and is not intended to monitor executing C programs. It may seem external variables would have no meaning if the monitor was run in the interpreter, but Copilot gives the user the ability to specify default stream values for an external variable that get used when the monitor interpreted.

A Copilot specification is *open* if defined with external symbols in the sense that values must be provided externally at runtime. To simplify writing Copilot specifications that can be interpreted and tested, constructs for external symbols take an optional environment for interpretation.

External variables are similar to global variables in other languages. They are defined by using the `extern` construct:

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

It takes the name of an external variable, a possible Haskell list to serve as the environment for the interpreter, and generates a stream by sampling the variable at each clock cycle. For example,

```
sumExterns :: Stream Word64
sumExterns = let ex1 = extern "e1" (Just [0..])
                 ex2 = extern "e2" Nothing
             in  ex1 + ex2
```

is a stream that takes two external variables `e1` and `e2` and adds them. The first external variable contains the infinite list `[0,1,2,...]` of values for use when interpreting a Copilot specification containing the stream. The other variable contains no environment (`sumExterns` must have an environment for both of its variables to be interpreted).

Sometimes, type inference cannot infer the type of an external variable. For example, in the stream definition

```
extEven :: Stream Bool
extEven = e0 `mod` 2 == 0
  where e0 = externW8 "x" Nothing
```

the type of `extern "x"` is ambiguous, since it cannot be inferred from a Boolean stream and we have not given an explicit type signature. 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.

In general it is best practice to define external symbols with top-level definitions, e.g.,

```
e0 :: Stream Word8
e0 = extern  "e0" (Just [2,4..])
```

so that the symbol name and its environment can be shared between streams.

Just like regular variables, arrays can be sampled as well. Copilot threats arrays in the same way as it does for scalars.

Example 6:
Lets take the example where we have the readouts of four pitot tubes, giving us the measured airspeed:

```
/* Array containing readouts of 4 pitot tubes. */
double airspeeds[4] = ... ;
```

In our Copilot specification, we need to provide the type of our array, because Copilot need to know the length of the array we refer to. Apart from that, referring to an external array is like referring to any other variable:

```
airspeeds :: Stream (Array 4 Double)
airspeeds = extern "airspeeds" Nothing
```

Triggers. Triggers, the only mechanism for Copilot streams to effect the outside world, 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 = externW64 "x" (Just [1..])
  trigger "h" (x < 10) [arg x]
```

The order in which the triggers are defined is irrelevant. To interpret this spec we run:

```
interpret 10 spec
```

which will yield the following output:

```
f:       g:        h:
(1,0)    ()        (1)
--       ()        (2)
(2,4)    ()        (3)
--       ()        (4)
```

```
(5,16)    ()        (5)
--        ()        (6)
(13,36)   --        (7)
--        --        (8)
(34,64)   --        (9)
--        --         --
```

Example 7:

We consider an engine controller with the following property: If the temperature rises more than 2.3 degrees within 0.2 seconds, then the fuel injector should not be running. Assuming that the global sample rate is 0.1 seconds, we can define a monitor that surveys the above property:

```
propTempRiseShutOff :: Spec
propTempRiseShutOff =
  trigger "over_temp_rise"
    (overTempRise && running) []

  where
  max = 500 -- maximum engine temperature

  temps :: Stream Float
  temps = [max, max, max] ++ temp

  temp = extern "temp" Nothing

  overTempRise :: Stream Bool
  overTempRise = drop 2 temps > (2.3 + temps)

  running :: Stream Bool
  running = extern "running" Nothing
```

Here, we assume that the external variable `temp` *denotes the temperature of the engine and the external variable* `running` *indicates whether the fuel injector 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. Notice there is a latency of one tick between when the property is violated and when the guard becomes true.*

4 Complete example

This section describes a complete use case example in Copilot. We will be using one of the provided examples as our code, and focus more on using Copilot within a project.

The code implements a simple heater, which turns on when the temperature drops below a certain point, and turns off if the temperature is too high. The temperature is read from a sensor

returns a byte, with a range of -50.0°C to 100.0°C. For easy of use, the monitor translates the byte to a float within this range.

4.1 C Code

Lets start of with the C program our monitor to connect to.

```
1  #include <stdlib.h>
2  #include <stdint.h>
3
4  #include "heater.h" /* Generated by our specification */
5
6  uint8_t temperature;
7
8  void heaton (float temp) {
9    /* Low-level code to turn heating on */
10 }
11
12 void heatoff (float temp) {
13   /* Low-level code to turn heating off */
14 }
15
16 int main (int argc, char *argv[]) {
17   for (;;) {
18     temperature = readbyte(); /* Dummy function to read a byte from a sensor. */
19
20     step();
21   }
22
23   return 0;
24 }
```

For this code we left out the low-level details for interfacing with our hardware. Let us look at a couple of interesting lines:

Line 4 Here we include the header file generated by our Copilot specification (see next section).

Line 8 Global variable that stores the raw output of the temperature sensor. This variable should be global, so it can be read from the code generate from our monitor.

Line 8-14 Functions that turn on and turn off the heater, low-level details are provided.

Line 17-21 Our infinite main-loop:

Line 18 Update our global temperature variable by reading it from the sensor.

Line 20 Execute a single evaluation step of Copilot. `step()` is imported from the `heater.h`, and is the only publicly available function from the specification.

As the code shows, the rate at which Copilot is updated is entirely up to the programmer of the main C program. In this case it is updated as quick as possible, but we could have opted to slow it down with a delay or a scheduler. Theoretically there could be multiple calls to `step()` throughout the program, but this complicated things and is highly discouraged.

4.2 Specification

The code for this specification can be found in the `Examples` directory of Copilot, or from the repository[4].

```
1  -- ------------------------------------------------------------------------
2  -- Copyright 2019 National Institute of Aerospace / Galois, Inc.
3  -- ------------------------------------------------------------------------
4
5  -- This is a simple example with basic usage. It implements a simple home
6  -- heating system: It heats when temp gets too low, and stops when it is high
7  -- enough. It read temperature as a byte (range -50C to 100C) and translates
8  -- this to Celcius.
9
10 module Heater where
11
12 import Language.Copilot
13 import Copilot.Compile.C99
14
15 import Prelude hiding ((>), (<), div)
16
17 -- External temperature as a byte, range of -50C to 100C
18 temp :: Stream Word8
19 temp = extern "temperature" Nothing
20
21 -- Calculate temperature in Celcius.
22 -- We need to cast the Word8 to a Float. Note that it is an unsafeCast, as there
23 -- is no direct relation between Word8 and Float.
24 ctemp :: Stream Float
25 ctemp = (unsafeCast temp) * (150.0 / 255.0) - 50.0
26
27 spec = do
28   -- Triggers that fire when the ctemp is too low or too high,
29   -- pass the current ctemp as an argument.
```

[4] https://github.com/Copilot-Language/Copilot/blob/master/Examples/Heater.hs

```
30    trigger "heaton"  (ctemp < 18.0) [arg ctemp]
31    trigger "heatoff" (ctemp > 21.0) [arg ctemp]
32
33  -- Compile the spec
34  main = reify spec >>= compile "heater"
```

The code should be pretty self explanatory. Note that we opted to use a `main`-function, which reifies and compiles the code for us.

On line 25 we can see the `ctemp`-stream, which is the temperature translated to Celcius. Interestingly, we need to do a manual typecast from `Word8` to `Float` using `unsafeCast`. This is a function provided by Copilot that can cast a stream to a different type in an unsafe manner, i.e. there may not be an exact representation of the value in both types. For this code, it might happen that an integer value cannot be represented exactly with a floating point.

4.3 Generating C code

Because we defined the `main`-function in our specification, generating code is now really easy:

```
$ runhaskell Heater.hs
```

This runs our Haskell code, with compiling a binary first. It runs that `main`-function and generates the C code of our monitor. Note that it called the files `heater.h` and `heater.c`, as defined by the prefix passed to the `compile`-function.

The next step is as easy as compiling our C code together with the monitor. We have opted to use GCC in a C99 mode with a extended set of warnings:

```
$ gcc -Wall -std=c99 heater.c main.c -o heater
```

If the implementations for `heaton`, `heatoff` and `readbyte` are provided somewhere, this should give us a nicely compiled binary called `heater`. Running this will provide us with a system that turns the heater on, when the temperature drops below 18.0°C and turns it off once it becomes higher then 21.0°C.

References

[BF93] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19:3–12, January 1993.

[DDE08] M.B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 228–237, 2008.

[Far04] H.A. Farhat. *Digital design and computer organization*. Number v. 1 in Digital Design and Computer Organization. CRC Press, 2004.

[Gil09] Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[GP10] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

[Sha01] Lui Sha. Using simplicity to control complexity. *IEEE Software*, pages 20–28, July/August 2001.