NASA/TM–2020–220587

Copilot 3

Ivan Perez
National Institute of Aerospace, Hampton, Virginia

Frank Dedden
Royal Netherlands Aerospace Center, Amsterdam, The Netherlands

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

April 2020

NASA/TM–2020–220587

Copilot 3

Ivan Perez
National Institute of Aerospace, Hampton, Virginia

Frank Dedden
Royal Netherlands Aerospace Center, Amsterdam, The Netherlands

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2020

Acknowledgments

The authors wish to thank Kaveh Darafsheh, Lee Pike, Robin Morisset, Sebastian Niller, Nis Wegmann, Jonathan Laurent, Matt Clark, Laura Titolo, Swee Balachandran, César Muñoz, Paul S. Miner, Chris Hathhorn, Eli Mendelson, Georges-Axel Jolayan, Macallan Cruff, Ryan Spring, and Lauren Pick.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

Ultra-critical systems require high-level assurance, which cannot always be guaranteed in compile time. The use of runtime verification (RV) enables monitoring these systems in runtime, to detect property violations early and limit their potential consequences. The introduction of monitors in ultra-critical systems poses a challenge, as failures and delays in the RV subsystem could affect other subsystems and threaten the mission as a whole. This paper presents Copilot 3, a runtime verification framework for real-time embedded systems. Copilot monitors are written in a compositional, stream-based language with support for a variety of Temporal Logics (TL), which results in robust, high-level specifications that are easier to understand than their traditional counterparts. The framework translates monitor specifications into C code with static memory requirements, which can be compiled to run on embedded hardware. This paper presents version 3 of the Copilot language, demonstrates its suitability with a number of examples, and discusses its use in larger applications. Additionally, it describes the framework's architecture, its implementation as a Domain Specific Language (DSL) embedded in Haskell, and the progress of the project over the years.

Contents

1 Introduction

Embedded systems are used in a wide range of applications, ranging from televisions and cellphones, to automobiles, aircraft and ships. In all of these applications, we want the system to function correctly, but those systems that are *safety critical*, where failure can result in injury or death of a human, warrant special attention [Knight, 2002]. To achieve the necessary level of reliability, both hardware and software of safety-critical systems need to be of very high quality.

Formal verification techniques are one method for achieving the level of reliability required in safety-critical systems. Generally, formal verification is based on mathematically proving correctness properties of a model of the system under study. Formal methods vary in the abstractions used to represent models, how faithful those models can be to the systems they represent, what properties can be captured and verified, how exhaustive proof can be, how efficiently proofs or counter examples can be produced, and how automated this process can be, among many other things.

Although there have been considerable advances in creating industrial-scale formal methods, it is not yet practical to apply them to an entire system. Formal verification is normally carried out on a model of the system rather than the software itself, and so the properties verified may not hold if the model is inaccurate or if other faults make the system behave unpredictably. Moreover, technological advances are enabling increasingly autonomous systems employing machine learning and similar methods that are not amenable to formal verification.

Runtime verification (RV) [Havelund and Goldberg, 2008, Goodloe and Pike, 2010, Bartocci et al., 2018] is a verification technique that has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. In RV, the system is monitored *during execution*, to detect and respond to property violations that take place during the actual mission. RV detects when properties are violated *in runtime*, so it is not a proof of correctness, but a significant improvement over testing alone.

Correct implementation of these monitors and the RV subsystem is crucial for the safe operation of the complete system and the success of the mission as a whole. The introduction of errors in the RV subsystem could disable or affect other subsystems, or lead to suboptimal deviations from the mission. In resource-constrained environments and time-critical systems, runtime monitors are commonly implemented in C for performance and memory constraints. This results in low-level code that is error prone, hard to understand and difficult to maintain.

In this paper we present Copilot 3, a runtime verification framework to write high-level specifications. Copilot is implemented as a stream-based, deeply embedded domain-specific language in Haskell. Streams are used to specify monitors, which denote functions that detect when properties are violated. Once a monitor is triggered, a user-defined function is called to take appropriate action. Our framework provides a constrained set of operations to define and combine streams, which guarantees that they are well-formed. The language also relies on dependent types, to enable safe use of non-primitive data structures, like structs and arrays. Copilot translates definitions into a subset of C99 with predictable memory requirements and real-time guarantees. To run the monitor on an embedded system, the generated C99 code can be compiled to the target platform and integrated into larger applications. Additional Copilot libraries extend the core language with higher-level constructs, like clock-based period and phase, Boyer-Moore majority voting, and temporal logic [Pnueli, 1977, Manna and Pnueli, 1992].

The contributions of this paper are as follows:

- We summarize the history of the Copilot project and discuss its progress and evolution over the years.
- We introduce the Copilot 3 specification language, which simplifies prior versions of the language [Pike et al., 2010, Pike et al., 2013] and extends it with notions of arrays and structs.
- We demonstrate how to specify runtime monitors using basic stream-level functions as well as different temporal logics.
- We discuss how we are using Copilot in larger applications built with NASA's Core Flight System (cFS).
- We describe the structure of the project, the architecture of the compiler, and its implementation as a domain-specific language in Haskell.
- We provide formal definitions of Copilot's grammar, typing rules, and big-step semantics.

Notation

Throughout the main chapters in this document, we generally use different fonts and symbols, with the following meanings. We use typescript (e.g., `release`) to represent a construct in the Copilot language, or a programming term in the host language Haskell. We use Greek lowercase letters (e.g., ϕ) to represent logical propositions. Symbols (e.g., $\Diamond$) and uppercase letters in calligraphic font (e.g., $\mathcal{U}$) denote logical operators. Italics are used mainly to emphasize text. The text **Example** at the beginning of a paragraph, and the symbol $\square$ aligned to the right margin, indicate the beginning and end of an example that illustrates the idea that precedes it.

2 Background

In September 2008, NASA awarded Galois, Inc. and the National Institute of Aerospace a contract "Monitor Synthesis for Software Health Management" (NNL08AA19B, order NNL08AD13T) to perform research in the area of runtime verification applied to hard real-time distributed avionics. The first deliverable was a comprehensive survey of the field of runtime verification [Goodloe and Pike, 2010], which found that most existing RV frameworks did not accommodate the very specialized domain of avionics, did not meet key constraints required for hard real-time, and/or required extensive instrumentation of the code being monitored.

Consequently, the PIs embarked on the development of a new RV framework, guided by an additional constraint: monitors for hard real-time avionics should not affect the system under observation in a way that changes the functionality of the system, requires re-certification, interferes with timing, or exhausts size, weight, and power (SWAP) reserves. To meet SWAP requirements, it was decided that monitors should be constant-time and constant-space code. The two most significant design decisions were: the RV framework that came to be called Copilot would be implemented as a Haskell embedded domain specific language (EDSL), and the specification language would be a stream-based data-flow language inspired by Lustre [Caspi et al., 1987] and LOLA [D'Angelo et al., 2005].

Early versions of Copilot compiled specifications into the Atom EDSL that, in turn, was used to generate monitors implemented in C. A beta release of Copilot and some experiments were described in [Pike et al., 2010]. The project showed enough promise to continue, and two more versions were released, maturing the specification language and the interpreter, as well as using SBV[1] as the C code generator. Copilot was demonstrated in numerous flight tests such as those described in [Pike et al., 2013]. The architecture of the Copilot 2.0 framework is described in [Pike et al., 2012] along with a description of how lightweight formal methods were applied to the problem of monitor correctness. To address the challenge of ensuring that a formal specification is correct, research was conducted integrating model checking and SMT capabilities into the Copilot 2.0 framework [Jonathan Laurent and Pike, 2015, Goodloe, 2016].

In spite of this progress, using Copilot 2.0 could be very clumsy in practice because, in embedded systems, many variables were stored as either C structs or C arrays, and Copilot could not handle these types of data structures. For instance, suppose we needed to monitor the position as provided by the avionics in terms of GPS coordinates, and it is stored in a C struct that gets updated every second. To monitor this information, we would have to write C code to unpack the structure into separate variables and make each one available to Copilot. Such situations were common and became an impediment to users. Given that neither Atom nor SBV supported generating code with arrays and structs, it was decided to build a new C code generator from scratch that could accommodate the needs of Copilot. This necessitated a considerable rewrite of the whole Copilot framework.

In addition to these revisions, the ability to invoke automatic theorem proving

[1]SMT Based Verification in Haskell: `https://hackage.haskell.org/package/sbv`.

tools such as model checkers and SMT solvers to prove properties about the Copilot specifications was removed. Although considered a useful feature of Copilot 2.0, it was removed in newer versions because the implementation depended on several Haskell open source packages that are no longer supported, and we lacked the resources to do all the work ourselves.

Copilot continues to be a research effort, and there are no plans to go through the FAA qualification process. Copilot is an open source development project with standard BSD license, so we invite any interested party to contribute.

3 The Copilot 3 Specification Language

Copilot is a framework that comprises an RV specification language, and a tool that compiles specifications into C code. Copilot specifications are defined by a series of *triggers*, that is, properties that need to be monitored paired with functions that need to be called when those properties become true. Properties themselves are defined as Boolean-carrying *streams*, using a rich language of stream definitions that includes primitives and combinators, and gives access to external streams defined in C.

3.1 Streams

Streams are infinite successions of values, and constitute the central entity of Copilot specifications. Streams can be defined using primitives, or be built from other streams with a series of combinators. We provide a limited Application Programming Interface (API) to ensure that, by construction, streams are well-formed and can be compiled to efficient C code.

3.1.1 Constant Streams

The simplest `Stream` definition in Copilot is a constant stream of values, for which we provide the primitive `constant` that takes an element and returns a stream consisting of that element at every sample.

Example The stream `true`, which Copilot defines in its Prelude for convenience, represents a constant stream carrying the Boolean value `True` and is defined as:

```
true = constant True
```

□

Because Copilot is a strongly typed domain-specific language, every expression and stream has a unique type. Following the notation of the host language Haskell, the type signature of every top-level definition is normally stated right before, prefaced by `::`. For example, `true`, as defined above, is a stream of Boolean values, which we denote with the type signature `true :: Stream Bool`.

The primitives and combinators that form the Copilot language are also functions that return streams. For example, the primitive `constant` itself is a Haskell function with type:

```
constant :: Typed a => a -> Stream a
```

The type signature of this primitive has two parts, separated by the symbol `=>`. On the right-hand side, the expression `a -> Stream a` indicates that `constant` is a function that takes an element of any type, which we call `a`, and returns a `Stream` carrying elements of the same type `a`. On the left-hand side, the expression `Typed a` is a *type constraint* and requires `a` to be an instance of the class `Typed`, denoting types that Copilot knows how to represent in C. We will not cover how to define custom types or instances in this document. Readers interested can consult standard Haskell textbooks to understand classes and instances, and the Copilot API to understand the type class `Typed`.

Including the type signatures explicitly is not always mandatory and it may be convenient to leave them out, especially when building very large and complex expressions. However, Copilot sometimes requires explicit type signatures for expressions that are ambiguous, in order to understand how to generate the corresponding C code. For example, the expression `constant 1` is a valid expression of type `Stream Int32`, but also one of type `Stream Int64` (and several other types). Without some type annotation, Copilot cannot know if it needs to use `uint32_t` or `uint64_t` in C to store the data, and so it requires that we spell out the type of the expression. To minimize the need for type annotations, Copilot provides a family of constant stream-building functions for each primitive type. For example, we can define the constant stream of 1's using 64-bit integer numbers as:

```
ones :: Stream Int64
ones = constI64 1
```

In this case, the type signature is redundant: because we have used the primitive `constI64`, the compiler knows we mean to build a stream of 64-bit integers.

3.1.2 Lifting and Point-wise Function Application

We provide definitions that extend the standard API of each type supported by Copilot to act pointwise on streams. Copilot supports Boolean values (i.e., `Bool`), signed and unsigned fixed-length integers (i.e., `Word8`, `Word16`, `Word32`, `Word64`, `Int8`, `Int16`, `Int32`, `Int64`), floating point numbers (i.e., `Float`, `Double`), limited structs, and limited arrays. The operators and combinators provided by Copilot are limited to a subset that we can compile to C efficiently. Details on structs and arrays are given in Sections 3.2 and 3.3 respectively.

Example The standard negation function `not`, operating on Booleans, would normally take one Boolean value and return another Boolean. Copilot defines `not` to operate on *streams of Booleans*. For example, the stream `false`, which holds the constant value `False`, could be defined by applying `not` to negate every value in the `true` stream defined earlier:

```
false :: Stream Bool
false = not true
```

Streams can contain other values representable in C, like integers and doubles. We overload literal numbers and mathematical operators to work on streams: literals denote constant streams, and operators are applied pointwise. For example, we can define the constant stream carrying the number 4 based on the definition of `ones` from before, as:

```
fours :: Stream Int64
fours = ones + 3
```

In this definition, the symbol `3` denotes the constant stream of `3`'s, and the symbol `+` denotes addition of streams carrying numbers, defined pointwise (e.g., the first element of `ones` plus the first element of `3`, the second element of `ones` plus the second element of `3`, and so on). □

3.1.3 Temporal Translations

Delaying a stream requires that we hold the stream's actual value in memory for future use. Unbounded delays (i.e., those in which the amount of elements to hold is potentially unbounded) are known to lead to memory leaks [Courtney and Elliott, 2001]. To make the generated C code efficient and memory usage predictable, we provide very limited ways of delaying streams: streams can be delayed by prepending *a fixed number of samples*, with the operator `(++)`, with type:

```
(++) :: Typed a => [a] -> Stream a -> Stream a
```

Example We can use the append operator `(++)` to create a stream that is initially `False` and later becomes `True` indefinitely:

```
falseThenTrue :: Stream Bool
falseThenTrue = [False] ++ true
```

Note that streams can be defined recursively. For example, we can define the stream that alternates between the values `True` and `False` as:

```
alternatingStream :: Stream Bool
alternatingStream = [True] ++ not alternatingStream
```

Using recursion, like before, we can define a step counter (e.g., $[1, 2, 3, \ldots]$) as follows:

```
counter :: Stream Int32
counter = [1] ++ (counter + 1)
```

□

Copilot also provides the opposite temporal transformation, dropping elements from a stream, with the function:

```
drop :: Typed a => Int -> Stream a -> Stream a
```

Example In a way similar to before, we can use drop to eliminate the first 5 elements from the stream `counter`, with the expression:

```
counterFromFive :: Stream Int32
counterFromFive = drop 5 counter
```

□

Dropping elements introduces a potential issue if the elements are not available, which may happen if they come from an external source (e.g., a sensor). This is discussed in the following.

3.1.4 External Streams

To connect Copilot specifications to existing C applications, we provide the primitive `extern` to define a stream based on the value of a global C variable, by indicating its name and its type. Within Copilot, we have no way of guaranteeing that the given variable exists, or that it has the expected type. However, from a specification containing an external stream, Copilot generates a C header file that declares the existence of an extern global variable with a specific type. The use of a variable name that does not exist, or that has the wrong type, would give rise to either an error or a warning when trying to compile and link the generated C code as part of a larger application.

Example Commonly in Copilot specifications, there is a need to access data provided by an external sensor. For example, given a global variable `pos_data`, of a type `Position`, holding the current position of a Unmanned Aerial Vehicle (UAV), we can use it in Copilot specifications as:

```
posdata :: Stream Position
posdata = extern "pos_data" Nothing
```

The additional argument `Nothing` contains an optional list of `Position`s that can be used during simulation, when actual data from the sensor is not available.

External streams are one example of a stream from which we cannot drop samples, since that would require being able to provide data that has not been produced by the system yet. If we try to drop samples from a stream, for example, by using the expression `drop 1 posdata` anywhere in our specification, Copilot reports a *compile-time* error:

```
Copilot error: Drop applied to non-append operation!
```

The error is not reported if we first append samples to the stream, for example, with `drop 1 ([p1, p2] ++ posdata)` (where `p1` and `p2` are two valid and known positions). If we drop more samples than we prepend, however, Copilot still reports an error:

```
Copilot error: Drop index overflow!
```

Problems with non-causal definitions are common in temporal frameworks [Bahr et al., 2015,Elliott and Hudak, 1997] and, in this way, Copilot makes some potential errors in specifications detectable at compile time.

□

3.2 Structs

A key contribution of this paper and Copilot 3 is the introduction of support for structs and arrays. Both are first-class citizens, i.e., they live at the same syntactical level as regular values like integers and Booleans.

Copilot structs are compiled into C structs and made available to the monitor. To be able to generate a correct `struct` definition in C, Copilot structs need to be defined using specific Copilot types. We normally implement structs in Copilot as records made of *fields*, each having a *name* and a *type*. We use the type `Field s t` to represent each field, with `s` being the field name (a type-level literal string), and `t` being the type of the field.

Example One of the properties we monitor in our systems is the temperature of the UAV's battery. The following defines a new Copilot type `Battery` with a field `temperature` of type `Int16`,[2] which corresponds to a struct in C with a field `temp` of type `int16_t`:

```
data Battery = Battery
  { temperature :: Field "temp" Int16 }
```

□

We provide a limited API to operate on streams of structs or records. Currently, Copilot supports projections, that is, accessing a field of a struct, with the function[3]:

```
(#) :: (Typed a, Typed t)
    => Stream a -> (a -> Field s t) -> Stream t
```

The first argument denotes the stream carrying a struct of type `a`, and the second denotes a field of the struct with name `s` and type `t`. Because structs are first class, they are valid types to be used in streams, and so are their fields.

Example If we have a global C variable `battery` of the struct type generated from the definition of `Battery` above, holding the state of the battery at each step, we can access it from Copilot with the following definition:

```
batt :: Stream Battery
batt = extern "battery" Nothing
```

We can extract a field of a stream of structs, producing another stream in a type-safe way:

```
tempPlus1 :: Stream Int16
tempPlus1 = batt # temperature + 1
```

□

3.3 Arrays

Copilot also includes support for arrays, with an advanced type that includes the length as part of the type of the array. For example, a stream in which each element is an array with 16 elements of type `Int64` would have type `Array 16 Int64`. The presence of the array's length as a type-level natural serves two purposes: first, it

[2]For the sake of simplicity, we omit other fields in the definition of `Battery`.

[3]The signature of `(#)` includes additional constraints. We omit details out of brevity, but this does not change the way that it is used.

allows the compiler to detect, at compile time, some incorrect accesses (i.e., access out of bounds), and, second, it allows us to generate C without dynamic memory allocation, as all arrays have known, fixed lengths. The details of how this is implemented are discussed in Section 6.

Similarly to structs, Copilot provides a limited API to work with `Stream`s of `Array`s. To access specific elements in the array, we provide the operator[4]:

```
(.!!) :: (KnownNat n, Typed t)
      => Stream (Array n t)
      -> Stream Word32
      -> Stream t
```

This operation allows us to access an element of an array, where the position of that element is determined by a number in a stream.

Example We can augment `Battery` with the measurements of the voltages of individual cells by including a new field:

```
data Battery = Battery
  { temperature :: Field "temp"  Int16
  , voltages    :: Field "volts" (Array 10 Word16)
  }
```

The field `voltages` is an array of length 10, whose elements have type `Word16`.

Copilot allows us to define streams that access specific elements of these arrays. For example, we can take the `voltages` field from `batt`, extract its first element, and add one to the result.

```
volt0Plus1 :: Stream Word16
volt0Plus1 = (batt # voltages .!! 0) + 1
```

Copilot is able to detect some incorrect accesses at compile time, if the index within the stream is out of range and the Copilot expression denoting that index is a constant stream. For example, if we had passed 11 as second argument of `(.!!)`, we would have seen a warning during compilation. Nevertheless, it is not generally possible for all streams to detect incorrect accesses in compile time, since the value of the stream containing the index may only be determined at runtime if they depend on some external variable.

□

3.4 Monitors

The purpose of Copilot is to monitor properties and to raise an alert when an assertion is violated. The Copilot language defines monitors as sequences of *triggers*. A trigger is defined as a stream of Booleans, a C function to be called when the current sample is true, and the arguments to pass to that function:

[4]The signature of `(.!!)` imposes additional constraints on the array, which we omit, but the arguments and the way they are used are as described.

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

`Spec` is a type internal to Copilot and represents a specification. `Spec`s are monadic computations, so we can declare multiple triggers by listing them in sequence in `do` notation.

Properties in triggers denote violations, not assertions. Therefore, triggers denote functions to call when samples are `True`, not `False`.

The function to call is given by the first argument as a `String`, and needs to be implemented by the user. Referring to a function that does not exist would lead to a linking error. If the header files generated by Copilot are included in other C files of the application that uses Copilot, referring to a function with the wrong arguments in a trigger would also lead to a compilation error.

Example The following specification declares two monitors. The first one executes the C function `large`, passing as argument the current value of the stream `counter`, when the voltage of the first cell of the battery, plus one unit, is greater than 8. The second monitor calls the function `too_large` with no arguments when the same voltage is greater than 10 (see Sec. 3.3):

```
monitor = do
  trigger "large" (volt0Plus1 > 8) [arg counter]
  trigger "too_large" (volt0Plus1 > 10) []
```

Copilot specifications can be simulated on a computer, or compiled into C code to be used in the same or a different device. We refer readers to Section 2 (*Interpreting and Compiling*) of the Copilot tutorial for up-to-date instructions on how to simulate and compile Copilot specifications using different backends available.

□

4 Logics and Languages

Monitors and specifications can become overly complex as systems grow. To aid understanding, Copilot supports the standard logic operators from propositional logic, as well as temporal combinators based on temporal logics.

4.1 Logical Operators

As mentioned in Section 3, Copilot extends the standard APIs of the supported types to apply pointwise on streams. In the case of Booleans, Copilot provides a number of logical operators based on propositional logic. Apart from the constants `true` and `false`, the following operators on Boolean streams are provided:

```
not   :: Stream Bool -> Stream Bool
(&&)  :: Stream Bool -> Stream Bool -> Stream Bool
(||)  :: Stream Bool -> Stream Bool -> Stream Bool
xor   :: Stream Bool -> Stream Bool -> Stream Bool
(==>) :: Stream Bool -> Stream Bool -> Stream Bool
```

In all cases, these operators apply the associated Boolean operation to the values at each sample. For example, given two Boolean streams `s1` and `s2`, the stream `s1 ==> s2` is true at a time (i.e., sample) if `s2` is true at that time, or if `s1` is false.

While these logical operators can help simplify basic expressions, the complexity of real-world applications demands higher-level languages. In the following we explore temporal logics supported by Copilot, and introduce operators to refer to past or future values of a stream.

4.2 Temporal Logics

Generally speaking, temporal logics extend other logics with a temporal dimension. To describe relations between formulas at different times, temporal logic languages introduce modal operators that abstract over time. For example, some languages provide an operator $\Box$ (called *always*, also written G, after the word *globally*), and a formula $\Box\, \phi$ is true if and only if ϕ is true at all times, where the specific meaning of the expression "at all times" depends on the logic.

Temporal logic languages generally vary in the logic they are based on, the temporal operators they support and in their model of time (e.g., continuous vs discrete, linear vs branching, future and/or past, etc.). These aspects impact what formulas can be expressed, which ones are true or false, what information is needed to evaluate them, and how efficiently we can do so.

Because time is an essential component of stream languages, like Copilot, temporal logics constitute a suitable mechanism to express many of the re-occurring patterns in monitor specifications. In the following we discuss some of the languages supported by Copilot, and demonstrate their use with examples.

4.2.1 Past-time Linear Temporal logic

Past-time Linear Temporal Logic (ptLTL) is an extension of propositional logic in which time is seen as linear, discrete, and bounded. While, in propositional logic, every variable may take the value true or false, in ptLTL, every variable may take the value true or false, *at each point in the present or in the past.*

Past-time linear temporal logic introduces temporal operators, letting us express logical formulas based not only on certain propositions being true at the present, but also on their validity in the past.

Copilot supports the temporal operators `alwaysBeen`, `eventuallyPrev`, `previous`, and `since`, all operating on and returning Boolean streams:

```
alwaysBeen     :: Stream Bool -> Stream Bool
eventuallyPrev :: Stream Bool -> Stream Bool
previous       :: Stream Bool -> Stream Bool
since          :: Stream Bool -> Stream Bool -> Stream Bool
```

The meaning of these operators is relatively simple. A stream `alwaysBeen x` is true at a time if `x` has been true at all times, present and past (Fig. 1). The operator `eventuallyPrev` works the opposite way, and `eventuallyPrev x` is true if `x` has ever been true. The temporal operator `previous` refers to the immediately previous sample, with `previous x` being true if `x` was true in the last sample. Finally, `since x y` is true at a time if the stream `x` has been continuously true since the first sample after `y` became true.

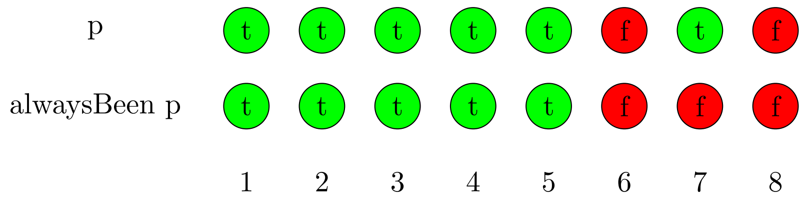


Figure 1: Example of values of the formula `alwaysBeen p` for different values of `p` at different times.

Example Borrowing the example in the prior section, imagine that we want to detect if the first voltage of the battery was ever too high. We can express that in Copilot with the following specification:

```
voltageWasTooHigh :: Stream Bool
voltageWasTooHigh = eventuallyPrev ((batt # voltages .!! 0) >= 10)
```

We can combine these temporal operators with the pointwise operators presented earlier in this section, to capture, for example, that a safety system can only be activated if a condition was violated before:

```
safetyResponseOK :: Stream Bool
safetyResponseOK = safetyEngaged ==> voltageWasTooHigh
  where
```

```
    safetyEngaged :: Stream Bool
    safetyEngaged = extern "safety_system" Nothing
```

$\square$

4.2.2 Bounded Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension of propositional logic in which time is seen as linear, discrete, and unbounded. Analogously to ptLTL, in LTL, every variable may take the value true or false, at each point in time, *present or future*.

LTL includes several operators that extend the logic with a temporal dimension and abstract over common temporal notions:[5] $\square$ (*always*), $\Diamond$ (*eventually*), $\bigcirc$ (*next*), $\mathcal{U}$ (*until*) and $\mathcal{R}$ (*release*). The first four operators are the future-oriented counterparts of the operators introduced for ptLTL. For example, the formula $\square\, \phi$ is true if ϕ is true at all times, present and future.

In LTL, it may not always be immediately possible to determine the validity of a formula when performing runtime verification. For example, if a stream `f` is false at a time, then `always f` is definitely false at all times, present and future. However, if a stream `f` is true at present, we cannot immediately determine the present value of `always f`, because there is a chance that `f` will become false at a future time.

Different forms of LTL adapt to this aspect differently. Some include a multi-valued logic, in which the value of a temporal proposition is not just true or false, but may also be "true so far" or "undecided". In Copilot, we opt for implementing Bounded LTL, a variant of LTL in which the amount of time into the future that is observable is bounded in each application of a temporal operator. For example, the expression `always n f` is true if `f` is true at all times from now until `n` samples from now. To be able to make a decision about the validity of a formula, our implementation of Bounded LTL requires that we have enough samples available in the stream (i.e., that we can drop samples from it). In particular, extern streams used in Bounded LTL properties must have additional samples prepended at the beginning with `(++)`.

The corresponding Copilot operators for Bounded LTL are as follows:

```
next       :: Stream Bool -> Stream Bool
eventually :: Integral a => a -> Stream Bool -> Stream Bool
always     :: Integral a => a -> Stream Bool -> Stream Bool
until      :: Integral a
           => a -> Stream Bool -> Stream Bool -> Stream Bool
release    :: Integral a
           => a -> Stream Bool -> Stream Bool -> Stream Bool
```

The constraint `Integral a` indicates that we can express the number of samples into the future that we want to observe using any type that is an integral number. All of these operations require the Boolean argument streams to have, at least, as many samples as the first parameter indicates, except for `next`, which requires that we can drop, at least, one sample.

[5]LTL has been extended with multiple operators. In the following, we present LTL with until and next, which includes all the operators supported by Copilot.

Example Continuing with the same example, we may be interested in verifying that the safety system is engaged in a timely fashion when a violation takes place.

We can capture part of that condition with the following Bounded LTL stream, which states that the safety system is engaged, at the latest, 5 steps after the violation is detected:

```
safetyResponseTimely :: Stream Bool
safetyResponseTimely = voltageWasTooHigh ==> eventually 5 safetyEngaged
```

Note that, for this to work, we would have to modify `safetyEngaged` to prepend at least four additional samples to it.

□

This implementation of Bounded LTL sits on top of Copilot, making the combination of the two potentially more expressive than the original logic. In particular, some properties are known not to be expressible in LTL, such as, for example, a formula that is true at every other sample, but they can be expressed in Copilot due to the existence of delays and recursion.

4.2.3 Metric Temporal Logic

Metric Temporal Logic (MTL) generalizes the notion introduced in Bounded LTL, and requires that every application of a temporal operator be accompanied by a range of time. For example, while in Bounded LTL, the formula $\Box_n\ \phi$ will be true if ϕ is true at all times from now until n samples from now, in MTL, the formula $\Box_{[n_1,n_2]}\ \phi$ will be true if ϕ is true at all times between n_1 and n_2. Since we can always make n_1 match the current or next time, we can easily define temporal operators from Bounded LTL in MTL.

Our implementation of MTL includes both future-facing and past-facing operators, and adds two additional parameters to every temporal operator for more versatility: first, a clock that indicates the current time, to facilitate re-sampling streams at different rates, and second, the minimum distance between two adjacent samples. For example, the signature of the MTL operator $\Box$ (*always*), analogous to the one for Linear Temporal Logic, is as follows:

```
always :: (Typed a, Integral a)  -- 'a' is the time and is an integral
                                 --     number representable in C
       => a                      -- ^ Lower time bound
       -> a                      -- ^ Upper time bound
       -> Stream a               -- ^ Clock stream
       -> a                      -- ^ Minimum time delta between samples
       -> Stream Bool
       -> Stream Bool
```

The first two arguments are times, and denote the bounds of the time period that is being considered (n_1 and n_2 in the example above). The next two denote the clock, which is a strictly positive signal carrying the current time, and a fixed number denoting the minimum time delta between two successive time samples.

Finally, the last argument, a Boolean stream, is the stream to which the temporal operator is actually being applied.

The types of other key MTL operators supported by Copilot are analogous and listed below, where the constraint `(Typed a, Integral a)` applies in every case but is omitted for brevity:

```
eventually     :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
eventuallyPrev :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
alwaysBeen     :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
until          :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
               -> Stream Bool
release        :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
               -> Stream Bool
since          :: a -> a -> Stream a -> a -> Stream Bool -> Stream Bool
               -> Stream Bool
```

Example A property that is commonly monitored in critical systems is whether some alert or property violation has been on for too long. That would generally indicate that the issue causing the alert may worsen, and/or that the systems that should have addressed it may be malfunctioning.

For example, the following property becomes true when the voltage remains too high for too many samples:

```
safetyResponseTimely' :: Stream Bool
safetyResponseTimely' = alwaysBeen 1 5 counter 1 voltageWasTooHigh
```

The first two arguments, `1` and `5`, indicate the time bounds for the application of the temporal operator (i.e., “between 5 samples ago and 1 sample ago”). The next two arguments, `counter` and `1`, are the stream that provides the current time, and the minimum distance between two successive times.

□

5 Applications

5.1 Copilot and NASA's Core Flight System

We have integrated Copilot monitors into applications built using Core Flight System (cFS) [Wilmot, 2005], an architecture for rapid integration of flight applications developed by the Flight Software Branch at NASA's Goddard Space Flight Center. In these applications, data and system updates from the flight computer of an aircraft are made available via communication channels using the MavLink protocol. Applications can be compiled to run on an embedded system (e.g., Arduino), or executed locally and visualized using a ground control system like MAVProxy (Fig. 2).

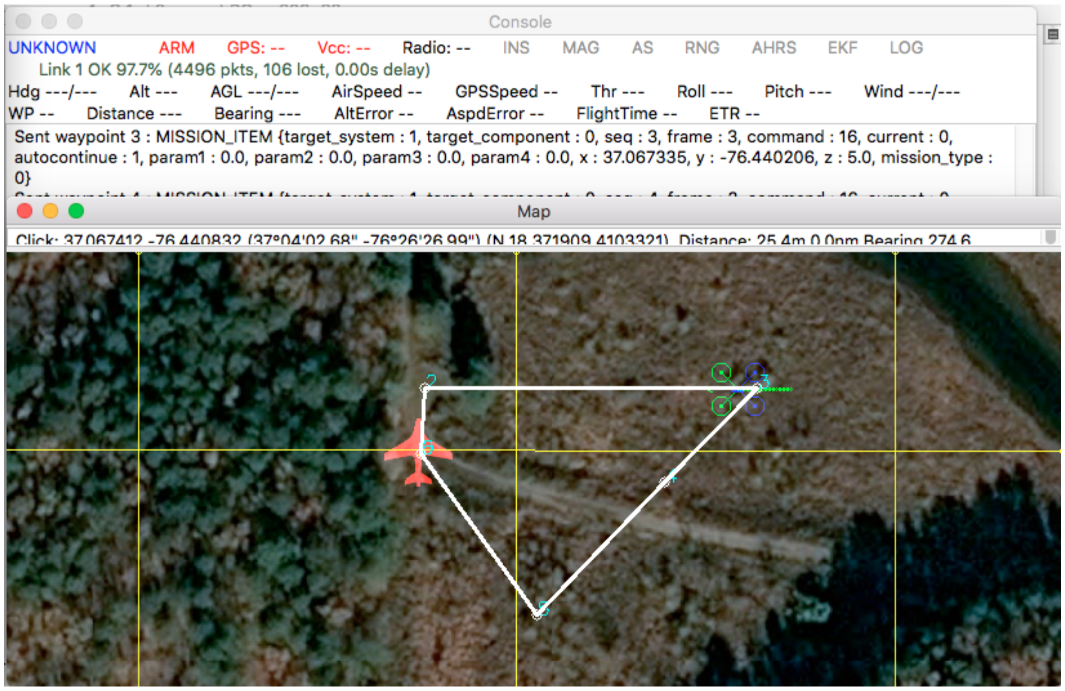
The integration of Copilot with cFS enables using Copilot as a Runtime Verification framework *for existing applications*. A Copilot cFS application subscribes to the right messages (*topics*), obtains the updates, and makes them available to monitors by copying the data to the global `extern` variables needed by the Copilot streams. This process is currently manual, and Copilot does not provide an automatic mechanism to generate cFS applications or copying data from all possible kind of messages (*topics*). After copying the data, the cFS application calls Copilot's main `step` function, which runs the monitors, detects property violations, and executes the trigger functions, if necessary. The same Copilot cFS application globally declares those functions needed by the Copilot triggers, which, at each sampling time, report property violations by writing messages to a log that can be observed during simulation or execution.

Generally, using structs and arrays simplifies monitors. Without them, each individual field would have to be made available as an individual stream, manually unpacking each structure in C to access it in the monitor.

We are currently in the process of running these monitors directly on a UAV in a controlled environment to evaluate their behavior in more realistic scenarios. We are also developing more advanced temporal monitors to validate the behaviour of ICAROUS (Independent Configurable Architecture for Reliable Operations of Unmanned Systems) [Consiglio et al., 2016], a software architecture for Unmanned Aerial Systems with a focus on safety and formal verification. ICAROUS is implemented using cFS and makes data available via MavLink with custom topics, making it simple to connect to Copilot (Fig. 2).

Monitoring ICAROUS requires the use of more abstract temporal logics, like Metric Temporal Logic. For example, one property we monitored during our initial development was whether traffic conflicts had been detected a few samples before ICAROUS was engaged. This property makes use of two external data sources. The first is used to determine if ICAROUS is engaged, and is made available by the field `acParam1` of the stream `argsCmd`. This field is of type `Double` and has value 1 if ICAROUS is engaged, and 0 otherwise:

```
icarousEngaged :: Stream Bool
icarousEngaged = (argsCmd # acParam1 == 1)
```



```
Console
UNKNOWN       ARM    GPS: --    Vcc: --    Radio: --    INS    MAG    AS    RNG    AHRS    EKF    LOG
    Link 1 OK 97.7% (4496 pkts, 106 lost, 0.00s delay)
Hdg ---/---    Alt ---    AGL ---/---    AirSpeed --    GPSSpeed --    Thr ---    Roll ---    Pitch ---    Wind ---/---
WP --    Distance ---    Bearing ---    AltError --    AspdError --    FlightTime --    ETR --
Sent waypoint 3 : MISSION_ITEM {target_system : 1, target_component : 0, seq : 3, frame : 3, command : 16, current : 0,
autocontinue : 1, param1 : 0.0, param2 : 0.0, param3 : 0.0, param4 : 0.0, x : 37.067335, y : -76.440206, z : 5.0, mission_type :
0}
Map
Click: 37.067412 -76.440832 (37°04'02.68" -76°26'26.99") (N 18 371909 4103321)  Distance: 25.4m 0.0nm Bearing 274.6
```

Figure 2: MAVProxy showing an aircraft (ownship) on the left and a potential intruder on the right. The system currently being executed runs both ICAROUS, as well as a Copilot cFS application that executes the monitors.

The stream `argsCmd` is itself defined globally (as `my_args` in the example below):

```
argsCmd :: Stream ArgsCmd
argsCmd = extern "my_args" Nothing
```

The type of this stream is a struct whose definition mirrors the one provided by ICAROUS in C. A simplified definition of the type `ArgsCmd` is:

```
data ArgsCmd = ArgsCmd
  { acTlmHeader :: Field "TlmHeader" (Array 16 Word8)
  , acParam1    :: Field "param1" Double
  }

instance Struct ArgsCmd where
  -- Name of the type in C
  typename _ = "argsCmd_t"
```

The second data source needed by the property we wish to monitor indicates the number of traffic conflicts at any given time, which is available in our Copilot specification as the field `bNumConflictTraffic` of the external stream `bands`. This external stream is, like before, a struct whose type mirrors the one defined by ICAROUS.

We monitor the aforementioned property using an MTL expression, which constrains the time when the collision alert should have been present to be between 5 and 3 samples prior to ICAROUS being engaged (Fig. 3). This property, used by the trigger, defines a violation, not an assertion:

```
icarousEngaged &&
  alwaysBeen 3 5 clock 1 (bands # bNumConflictTraffic == 0)
```

When the complete version of this Copilot specification, including the trigger definitions, is compiled and executed, it generates C code and header files that can be integrated as part of a larger cFS application.

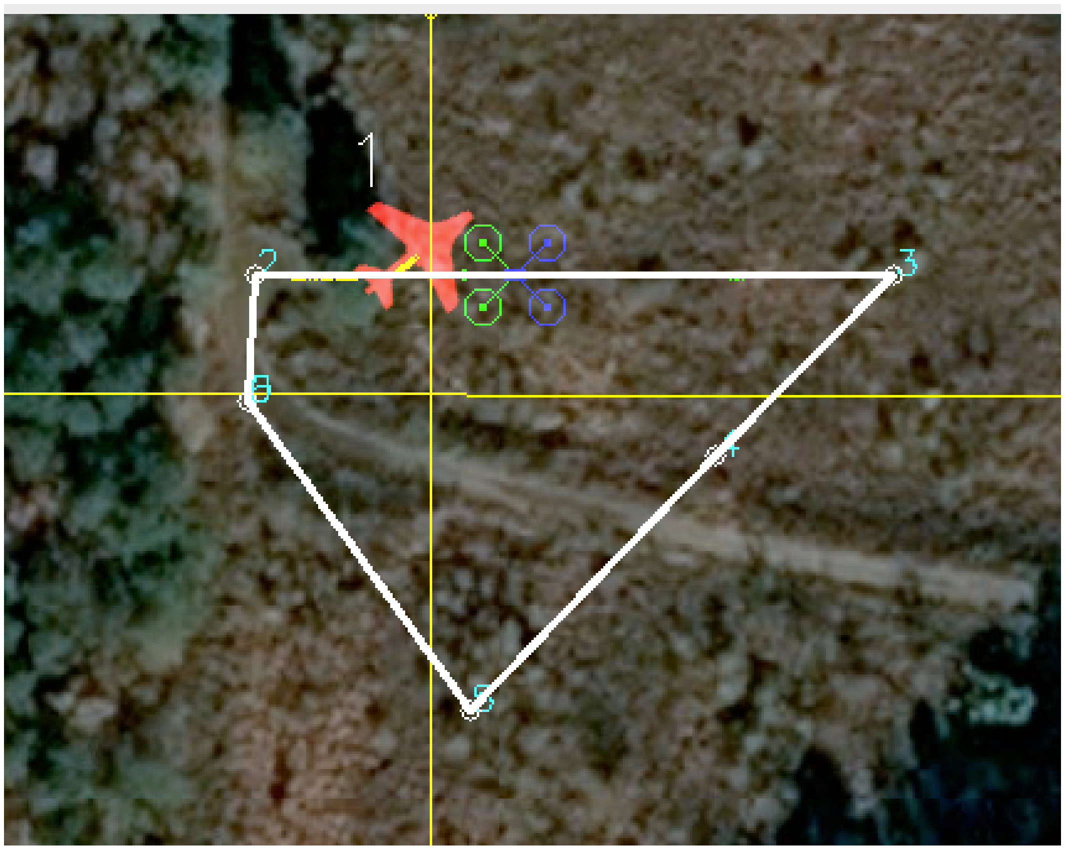


Figure 3: Detail of MAVProxy showing an aircraft, maneuvering to avoid a collision with an intruder. Copilot-generated C code is used by a cFS application to monitor the collision avoidance logic and report property violations.

6 The Architecture of Copilot 3

As described earlier, Copilot is used to write high-level specifications, which in turn are translated into C. With the widespread availability of C, this allows the code to be executed on virtually any platform.

6.1 Structure of the Copilot Project

Copilot specifications are written using the *Copilot Language*, which contains the EDSL described in Sec. 3. *Copilot Libraries* is an extra set of utilitarian functions that provide more high-level concepts on top of the language. For example, these libraries implement the temporal logic operators (Sec. 4), voting algorithms, support for clocks (i.e., Boolean streams with a specific frequency) and take / drop operators.

High-level Copilot specifications can be compiled into C99, or interpreted on the development machine. The behaviour of the C99 code and the interpreter should be operationally equivalent, except for the effects of the trigger functions themselves.

To help gain confidence on the correctness of the back-end, a testing suite using QuickCheck [Claessen and Hughes, 2011] has been implemented. QuickCheck generates a large number of random specifications, and the output of the execution of the generated C code is compared with the output of the interpreter. The relation between these components is captured in Figure 4.

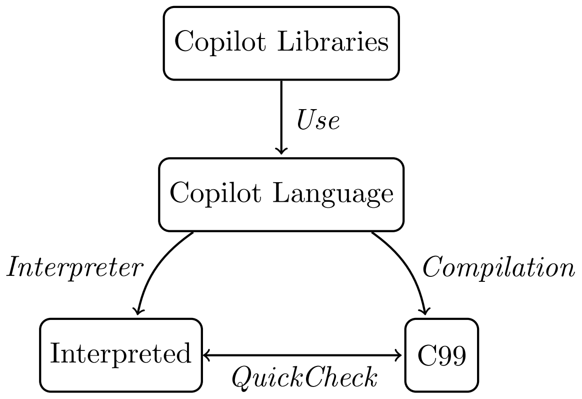


Figure 4: Copilot Project Structure

There are several other libraries as part of the Copilot project, some of which are used only internally. In the following sections we describe how the compilation to C code is performed by the user, and we later describe internal aspects of the compiler and other libraries involved.

6.2 Compiler Frontend

Copilot is implemented as a Domain Specific Language embedded in Haskell, and the compilation into C code is a two-step process (Fig. 5). First, the Copilot specification, written in Haskell, is *compiled* into an executable program. Second, the generated program is *executed*, which generates C code in a `.c` file and a `.h` header file. The benefit of this two-step process is that some language features can be captured at type level, leveraging static analysis on Haskell's type checker, and both the resulting Haskell program and the C compiler will perform additional static checks.

Copilot itself does not generate a complete C program that users can run directly. It is up to the user to provide a `main` function and the necessary variables to monitor. This approach enhances portability, as hardware-dependent parts can be written in C.

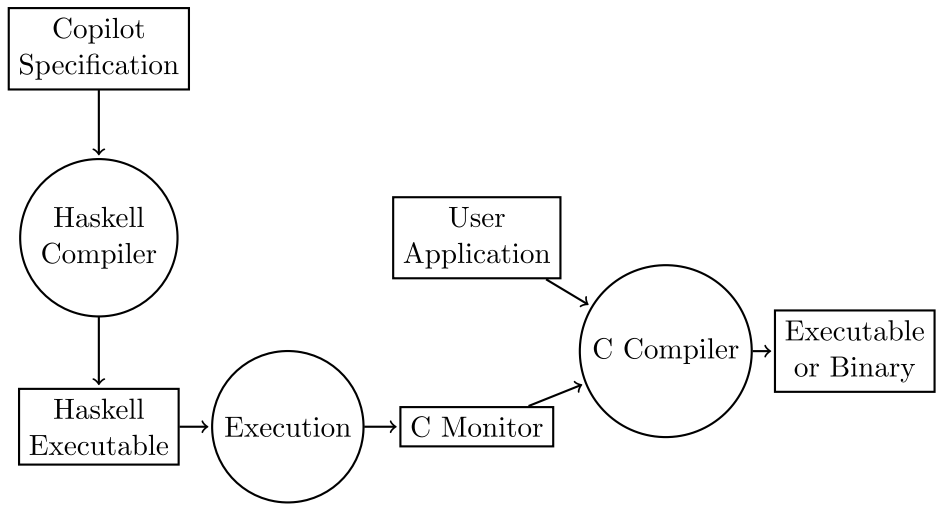


Figure 5: Compilation of specifications into C by first compiling Copilot specification to obtain an executable, and running the executable to obtain C code. The generated C code can be included and compiled with a C compiler as part of a larger application.

It is possible to convert the Copilot specification into C code in one step, by running the Copilot specification through a Haskell interpreter.

6.3 Compiler Backend

Copilot specifications can be written using the Copilot language, as well as one of the additional libraries provided, as described earlier in this section. Internally, high-level Copilot specifications are type-checked and compiled into *Copilot Core*, a relatively low-level specification language more suitable for compiling. Copilot Core specifications can be compiled into target code, or interpreted on the development machine (Fig. 6).

The specification provided by the user has the type `Language.Spec`, which needs to be compiled into C code. We do so in a number of steps, spanning several modules.

First, the specification is analysed for some common pitfalls, like dropping more elements than available in the buffer, and multiple definitions of the same trigger. Next, the `Language.Spec` is turned into a `Core.Spec` by a reification process. During this process, streams are given unique ids, so a form of sharing can be implemented. Additionally, the datatype implementing `Streams` in `Core` is simplified

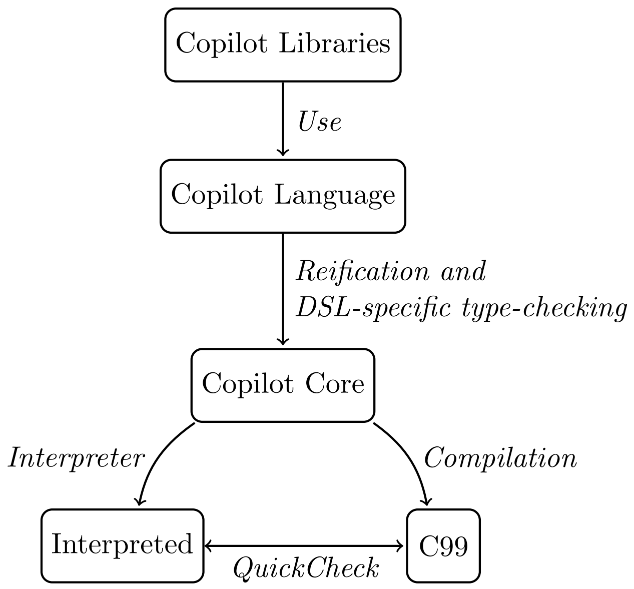


Figure 6: Copilot's architecture

and modified, facilitating compilation to C99.[6]

Compilation into C code is split in two parts: compiling the header file and compiling the monitors' implementations (Fig. 7).

Header file
Type definitions
Extern variables
Trigger declarations
Step declaration

C code file
Imports
Global variables
Stream generators
Step function

Figure 7: Parts of the generated header (`.h`) and C code (`.c`) files.

Compiling the header file is done by exploring the internal specification and obtaining all the external variables, their types, and the definitions of those types. In addition, the generated header file contains the declaration of all triggers, as well the main step function.

The generated C files are split in four main parts: imports, global variables, stream generators, and a main step function.

The generated C files make use of standard libraries. The header files `stdlib.h` and `stdint.h` are included by default, and so is the header file generated from the Copilot specification being compiled.

Next, the file includes global variables defining buffers needed to implement delay

[6]Although Copilot is designed to support multiple back-ends, outdated back-ends have been removed and currently only C99 is supported.

operations, as well as the current positions within each buffer. At the beginning, the buffer contains exactly the elements prepended to the stream, and, as the execution progresses, new elements from the delayed stream are written into this buffer.

Third, every stream is translated into a function that returns the current sample for that stream. A stream's current value is not held in a variable, but calculated each time in terms of the (current) values of other streams, by calling their associated sample-generation functions. For example, if we define a stream f as the pointwise addition of two streams g and h, the implementation of the function that returns the current sample for f will call the functions that return the current samples of g and h respectively, add them, and return that result. As a consequence of this design decision, the resulting C code describes data relations, akin to a data-flow graph, as opposed to describing how changes to external variables propagate forward to all the streams that use them. This is one of the key aspects of Copilot which best demonstrates its functional nature, and it is characteristic of fields like reactive programming and functional reactive programming.

Finally, the C code includes a main step function, which executes the monitors and advances the state of the Runtime Verification system in four steps: first, it copies all external variables; second, it checks all triggers, executing them if necessary; third, it updates stream buffers with new values, and finally, it increments stream indices. Copying of the external variables is necessary to make the code *reentrant*: it can continue its original execution after being interrupted. If this was not done, the execution of a monitor might be interrupted and resumed after variables used by the monitor have been updated. This can cause the same conceptual value to be different *within* one execution step, breaking the assumption of causality, which is disastrous from the point of view of reliability.

Note that this module does not generate a complete C program (i.e., it contains no `main` function). This approach helps integrate Copilot specifications into larger applications, as was described in Sec. 5.

We opted for C99 as our target language, which provides enough features of C to write clear output code, and is well-supported by a broad range of compilers, including GCC[7] and CompCert[8], a formally verified C compiler. To aid in writing the C code, Copilot makes use of a library implementing the C99 syntax. When Copilot's C backend was initially implemented, we could not find any libraries in Haskell that implemented the C99 standard precisely: while the formal C99 grammar is optimized for LR parsing, the libraries we found implemented a more abstract grammar that resulted in simpler constructs. Because our goal is to produce verifiable code, crucial in our domain of interest, we needed to stick to a representation that followed the standard exactly. Consequently, we implemented our own C99 library that strictly follows the syntax as defined in the standard.

[7]`https://gcc.gnu.org/`

[8]`http://compcert.inria.fr/`

7 Conclusion

In this paper, we described Copilot 3, a runtime verification framework for safety-critical, real-time embedded systems. We discussed the evolution of the project, how it was originally built, and how the components that make the framework have changed over time based on both project need and resources available. The new version of the language was introduced, and we saw how the new constructs of structs and arrays help to deal with more complex data structures without sacrificing safety. Copilot is a project rich in libraries, and we discussed different temporal logics supported by the language. We also discussed how the language is being used in combination with NASA Core Flight System applications, to add runtime verification capabilities to autonomous vehicles with minimal intrusion.

There are a number of open problems in this domain, and in Copilot as a whole, that remain to be addressed. We expect future versions of the language to be simpler and require less boilerplate code, and the frontend language, presented in this document, to be closer to the low-level core language used internally by Copilot. Finding the logic that is most appropriate to express the properties of autonomous vehicles remains an open problem in the runtime verification community. The use of Signal Temporal Logic [Maler and Nickovic, 2004], as well as new higher-level logics, could facilitate more consistent usage by domain experts.

Other stream-based languages and temporal formalisms, like Functional Reactive Programming [Elliott and Hudak, 1997] and Monadic Stream Functions [Perez et al., 2016], have recently seen advances that make them better suited for simulation, interactivity, fault tolerance, testing, and compilation to FPGAs [Finkbeiner et al., 2019,Perez, 2018,Perez et al., 2019]. The creation of a common abstraction that can be used to express temporal behaviour while facilitating reuse for multiple objectives is currently under study.

References

Bahr et al., 2015. Bahr, P., Berthold, J., and Elsman, M. (2015). Certified symbolic management of financial multi-party contracts. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 315–327, New York, NY, USA. ACM.

Bartocci et al., 2018. Bartocci, E., Falcone, Y., Francalanza, A., and Reger, G. (2018). Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer.

Caspi et al., 1987. Caspi, P., Pialiud, D., Halbwachs, N., and Plaice, J. (1987). LUSTRE: a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188.

Claessen and Hughes, 2011. Claessen, K. and Hughes, J. (2011). Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64.

Consiglio et al., 2016. Consiglio, M., Muñoz, C., Hagen, G., Narkawicz, A., and Balachandran, S. (2016). Icarous: Integrated configurable algorithms for reliable operations of unmanned systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–5. IEEE.

Courtney and Elliott, 2001. Courtney, A. and Elliott, C. (2001). Genuinely Functional User Interfaces. In *Haskell Workshop*, pages 41–69.

Damas and Milner, 1982. Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA. Association for Computing Machinery.

D'Angelo et al., 2005. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Manna, Z., Finkbeiner, B., Spima, H., and Mehrotra, S. (2005). LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE.

Elliott and Hudak, 1997. Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273. ACM.

Finkbeiner et al., 2019. Finkbeiner, B., Klein, F., Piskac, R., and Santolucito, M. (2019). Synthesizing functional reactive programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 162–175, New York, NY, USA. Association for Computing Machinery.

Goodloe, 2016. Goodloe, A. (2016). Challenges in high-assurance runtime verification. In *Leveraging Applications of Formal Methods, Verification and Validation:*

Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, pages 446–460.

Goodloe and Pike, 2010. Goodloe, A. and Pike, L. (2010). Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center.

Hall et al., 1994. Hall, C., Hammond, K., Jones, S. P., and Wadler, P. (1994). Type classes in haskell. In Sannella, D., editor, *Programming Languages and Systems — ESOP '94*, pages 241–256, Berlin, Heidelberg. Springer Berlin Heidelberg.

Havelund and Goldberg, 2008. Havelund, K. and Goldberg, A. (2008). *Verify Your Runs*, pages 374–383. Springer Berlin Heidelberg, Berlin, Heidelberg.

Jonathan Laurent and Pike, 2015. Jonathan Laurent, A. G. and Pike, L. (2015). Assuring the guardians. In *Proceedings of the 6th Intl. Conference on Runtime Verification*, LNCS 9333. Springer.

Knight, 2002. Knight, J. C. (2002). Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550. ACM.

Maler and Nickovic, 2004. Maler, O. and Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In Lakhnech, Y. and Yovine, S., editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg. Springer Berlin Heidelberg.

Manna and Pnueli, 1992. Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg.

Marlow, 2011. Marlow, S. (2011). Haskell 2010 language report. Technical report.

Perez, 2018. Perez, I. (2018). Fault tolerant functional reactive programming (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP).

Perez et al., 2016. Perez, I., Bärenz, M., and Nilsson, H. (2016). Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA. Association for Computing Machinery.

Perez et al., 2019. Perez, I., Goodloe, A., and Edmonson, W. (2019). Fault-tolerant swarms. In *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 47–54.

Pike et al., 2010. Pike, L., Goodloe, A., Morisset, R., and Niller, S. (2010). Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer.

Pike et al., 2012. Pike, L., Wegmann, N., Niller, S., and Goodloe, A. (2012). Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. ACM.

Pike et al., 2013. Pike, L., Wegmann, N., Niller, S., and Goodloe, A. (2013). Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4).

Pnueli, 1977. Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA. IEEE Computer Society.

Wilmot, 2005. Wilmot, J. (2005). A core flight software system. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 13–14, New York, NY, USA. ACM.

Appendix A

Grammar

The following describes the grammar of Copilot *expressions.* For both non-terminal symbols and terminal symbols, we use a notation based on Extended Backus-Naur Form. To facilitate reading and to avoid confusion between meta-level language (grammars) and language-level symbols (Copilot's), we use different fonts for different parts. When describing non-terminals, we use the notation $\langle \textit{symbol} \rangle$ to refer to non-terminal symbols, and the notation $\langle \textsc{symbol} \rangle$ to refer to terminal symbols. When the terminals expand to only one literal, for readability, we simply include the expected word in `monospace font`. Particular attention may be granted to the rule that allows expressions in Copilot to be parenthesized, in which parentheses in the Copilot language appear in monospace font, as opposed to, for example, the rule for $\langle \textit{valueList} \rangle$, in which the parentheses on the right-hand side indicate that the $*$ applies to everything inside the parentheses. The syntax $[a-z]$ indicates any characters between a lowercase a and a z. As standard in EBNF, the symbols $^?$, * and $^+$ denote meta-level postfix operators and indicate, that the term immediately preceding the symbol may, respectively, be optional, appear multiple times or not at all, and appear multiple times but at least once. A multiplicity operator, when subscripted by a numeric constraint, indicates a limit on the number of repetitions (e.g., $\langle A \rangle^*_{\leqslant 30}$ indicates that $\langle A \rangle$ can appear zero, one, or any number of times, up to 30 times). Expressions separated by a vertical line indicate choice (e.g., $\langle A \rangle | \langle B \rangle$ indicates that the expression is either $\langle A \rangle$ or $\langle B \rangle$). Extra spaces between tokens are ignored. There are terminal symbols whose definitions overlap. For example, the pattern for an $\langle \textsc{Id} \rangle$ matches also keywords in Copilot and in the host language. Generally, we assume terminals to be listed in order of precedence, and no Haskell keyword is a valid $\langle \textsc{Id} \rangle$ or $\langle \textsc{UId} \rangle$.

Because Copilot is a language embedded in Haskell, there are a number of connections between this grammar and the grammar of Haskell [Marlow, 2011]. First, the language generated by $\langle \textit{def} \rangle$ is contained by the language generated by the non-terminal $\langle \textit{decl} \rangle$ in Haskell's grammar (in particular, it is contained in the second case of the production, expanding to $\langle \textit{funlhs} \rangle\langle \textit{rhs} \rangle$). Also, the terminal symbols $\langle \textsc{Id} \rangle$ and $\langle \textsc{UId} \rangle$ correspond to the Haskell grammar elements $\langle \textit{varid} \rangle$ and $\langle \textit{consid} \rangle$, respectively. Second, some elements in Haskell that are needed or can be used by Copilot modules escape the description below. For example, we do not describe a mechanism to define new types or type classes, but assume that it exists in the host language and is correct. Generally, we have defined a minimal Copilot grammar, but any Haskell expression or type that reduces to a valid Copilot expression (value or type) will be accepted by the compiler.

A.1 Non-terminal Symbols

$$
\begin{array}{rcl}
\langle \textit{defs} \rangle & \rightarrow & \langle \textit{def} \rangle^{*} \\
\langle \textit{def} \rangle & \rightarrow & \langle \textsc{Id} \rangle \; \texttt{=} \; \langle \textit{stream} \rangle \\
\\
\langle \textit{stream} \rangle & \rightarrow & \texttt{(} \; \langle \textit{stream} \rangle \; \texttt{)} \\
 & | & \texttt{constant} \; \langle \textit{value} \rangle \\
 & | & \texttt{extern} \; \langle \textsc{String} \rangle \; \langle \textit{sampleV} \rangle \\
 & | & \langle \textit{valueList} \rangle \; \texttt{++} \; \langle \textit{stream} \rangle \\
 & | & \texttt{drop} \; \langle \text{V}\textsc{Int} \rangle \; \langle \textit{stream} \rangle \\
 & | & \langle \textsc{Op}1 \rangle \; \langle \textit{stream} \rangle \\
 & | & \langle \textit{stream} \rangle \; \langle \textsc{Op}2 \rangle \; \langle \textit{stream} \rangle \\
 & | & \langle \textsc{Op}3 \rangle \; \langle \textit{stream} \rangle \; \langle \textit{stream} \rangle \; \langle \textit{stream} \rangle \\
 & | & \langle \textit{stream} \rangle \; \texttt{\#} \; \langle \textsc{Id} \rangle \\
\\
\langle \textit{sampleV} \rangle & \rightarrow & \texttt{Nothing} \\
 & | & \texttt{(} \; \texttt{Just} \; \langle \textit{valueList} \rangle \; \texttt{)} \\
\langle \textit{valueList} \rangle & \rightarrow & [(\langle \textit{value} \rangle \; \texttt{,})^{*} \; \langle \textit{value} \rangle] \\
\\
\langle \textit{value} \rangle & \rightarrow & \langle \text{V}\textsc{Bool} \rangle \\
 & | & \langle \text{V}\textsc{Float} \rangle \\
 & | & \langle \text{V}\textsc{Int} \rangle \\
 & | & \texttt{array} \; \langle \textit{valueList} \rangle \\
 & | & \langle \text{U}\textsc{Id} \rangle \; \langle \textit{field} \rangle^{*} \\
\\
\langle \textit{field} \rangle & \rightarrow & \texttt{(} \; \texttt{Field} \; \langle \textit{value} \rangle \; \texttt{)}
\end{array}
$$

A.2 Terminal Symbols

$$
\begin{array}{rcl}
\langle\textsc{Op1}\rangle & \rightarrow & \texttt{not} \mid \texttt{abs} \mid \texttt{signum} \mid \texttt{complement} \\
& \mid & \texttt{recip} \mid \texttt{exp} \mid \texttt{sqrt} \\
& \mid & \texttt{log} \mid \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \\
& \mid & \texttt{asin} \mid \texttt{acos} \mid \texttt{atan} \\
& \mid & \texttt{sinh} \mid \texttt{cosh} \mid \texttt{tanh} \\
& \mid & \texttt{asinh} \mid \texttt{acosh} \mid \texttt{atanh} \\
& \mid & \texttt{cast} \mid \texttt{unsafeCast}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{Op2}\rangle & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`mod`} \mid \texttt{`div`} \\
& \mid & \texttt{/} \mid \texttt{**} \mid \texttt{`logBase`} \\
& \mid & \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{/=} \mid \texttt{>=} \mid \texttt{>} \\
& \mid & \texttt{||} \mid \texttt{\&\&} \mid \texttt{`xor`} \mid \texttt{==>} \\
& \mid & \texttt{.\&.} \mid \texttt{.|.} \mid \texttt{.\^{}.} \mid \texttt{.>>.} \mid \texttt{.<<.} \\
& \mid & \texttt{.!!}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{Op3}\rangle & \rightarrow & \texttt{mux}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{VBool}\rangle & \rightarrow & \texttt{true} \\
& \mid & \texttt{false}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{VInt}\rangle & \rightarrow & \texttt{-}^{?}[\mathsf{0}-\mathsf{9}]^{+}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{VFloat}\rangle & \rightarrow & \texttt{-}^{?}[\mathsf{0}-\mathsf{9}]^{+}\texttt{.}[\mathsf{0}-\mathsf{9}]^{+}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{String}\rangle & \rightarrow & \texttt{"}[\mathsf{a}-\mathsf{z}|\mathsf{A}-\mathsf{Z}][\mathsf{a}-\mathsf{z}|\mathsf{A}-\mathsf{Z}|\_|\mathsf{0}-\mathsf{9}]^{*}_{\leqslant 30}\texttt{"}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{Id}\rangle & \rightarrow & [\mathsf{a}-\mathsf{z}][\mathsf{a}-\mathsf{z}|\mathsf{A}-\mathsf{Z}|\mathsf{0}-\mathsf{9}|\_|\texttt{'}]^{*}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\textsc{UId}\rangle & \rightarrow & [\mathsf{A}-\mathsf{Z}][\mathsf{a}-\mathsf{z}|\mathsf{A}-\mathsf{Z}|\mathsf{0}-\mathsf{9}|\_|\texttt{'}]^{*}
\end{array}
$$

Appendix B

Typing Rules

In the following we list the typing rules of Copilot's high-level language. We limit ourselves to Copilot *stream* expressions formed from the grammar given previously, and do not include any aspects of the host language Haskell.

The typing domain is that of dependent types. In particular, the type of streams includes the amount of "lookahead" available in the streams. We do that to prevent dropping from external streams without appending enough samples. Dependently typed Copilot expressions can be assigned a standard Haskell type trivially, by dropping the values at type-level. This is illustrated at the end of this section, and the resulting types coincide with those available in Copilot.

We only give types to expressions formed from the non-terminal symbol $\langle \mathit{stream} \rangle$, and any symbol used, directly or indirectly, by it, and we do not include a typing rule for the non-terminals $\langle \mathit{defs} \rangle$ or $\langle \mathit{def} \rangle$. In accordance with our grammar, which does not include a way to define new Haskell datatypes, we assume the existence of the appropriate constructor and projection functions for structs. These are denoted at type-level with the symbol `->`, which has the common interpretation in Haskell of denoting a function. The definition of structs as Haskell records, standard in Copilot and described in Section 3, immediately provides the necessary projection functions and constructors as required by these typing rules.

The format for these rules follows that of [Damas and Milner, 1982], and use the notion of type classes existing in Haskell, which captures the idea of overloadable interfaces [Hall et al., 1994]. This notion is represented in these rules with the syntax $C\ k => \alpha$, which means that some type k, presumably mentioned in α, is an instance of some type class C. For example, the constraint $\mathit{Num}\ a$ indicates that the type a must be a type that is a number, like `Integer` or `Float`, and can not be, for example, `Bool`. The type classes used by these rules and their standard meaning is inherited from Haskell, and the requirements for a type to be an instance of each class can be consulted in the standard Haskell documentation.

It may come to the reader's attention that these rules include projections for structs and arrays, but no injection counterparts. In its present version, Copilot does not include functions to update elements of arrays or fields of structs.

To simplify rules, we abuse notation in streamOp1Floating with the use of `exp`, . . . , `atanh` to indicate "any operator in the production for $\langle$OP1$\rangle$ from the appearance of `exp` until `atanh`, both included". We also use the syntax `k` $\geqslant 0$ to indicate that k is a token whose numeric value is greater than or equal to zero. We use parts of the grammar used for streams to refer to elements that also exist at type-level, like numbers and strings.

In the rules that follow, the type for streams is a dependent type that includes the amount of lookahead available in the stream as part of its type. For example, a pure stream that does not depend on external data, like a constant stream, has infinite lookahead available: we may drop as many elements as we desire, and we can still obtain a valid element at the current time. In contrast, an external stream does not have any lookahead available: the only value available is the current one, and we cannot drop any elements from that stream (unless we concatenate elements first). We capture this notion in the types with a type-level number indicating the lookahead available as part of the type of streams `Stream` κ τ, with κ being the amount of lookahead and τ being the type of the values in the stream. The typing rules ensure that dropping elements is only well-typed for streams with values available. We use the notation $[\tau]_\kappa$ to refer to lists of known length κ containing elements of type τ. These features are only present in the types in these typing rules: from the user side, the only dependently-typed features available in Copilot are those discussed in Section 3.

The rules for casting between types are most conveniently expressed with two auxiliary relations that capture when casting is possible. We represent the safe casting relation with the function Ψ which, for any type, returns the countable set of types to which we can cast it safely, and we assume it possible to test for membership in that set (e.g., from `Int8` to `Int16`). Analogously, we use the relation Φ to represent the function for the types onto which casting is possible, but not safe (e.g., from `Int16` to `Int8`). The contents of these two relations are listed later in this section, in Tables B1 and B2. We also distinguish in these rules between the different implementations of `cast` and `unsafeCast` available by means of a sub-index, indicating the return type after casting.

B.1 Dependently Typed Typing Rules

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa\ \tau}{\Gamma \vdash \texttt{(}x\texttt{)} : \texttt{Stream}\ \kappa\ \tau} \qquad (\textsc{streamPar})$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{constant}\ x : \texttt{Stream}\ \infty\ \tau} \qquad (\textsc{streamConst})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash v : \texttt{Maybe}\ \texttt{[}\tau\texttt{]}_\kappa}{\Gamma \vdash \texttt{extern}\ s\ v : \texttt{Stream}\ 0\ \tau} \qquad (\textsc{streamExt})$$

$$\frac{\Gamma \vdash ls : [\tau]_{\kappa_1} \qquad \Gamma \vdash s : \texttt{Stream}\ \kappa_2\ \tau}{\Gamma \vdash ls\ \texttt{++}\ s : \texttt{Stream}\ (\kappa_1 + \kappa_2)\ \tau} \quad (\textsc{streamAppend})$$

$$\frac{\Gamma \vdash i : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{Stream}\ \kappa\ \tau \qquad i \leqslant \kappa}{\Gamma \vdash \texttt{drop}\ \ i\ x : \texttt{Stream}\ (\kappa - i)\ \tau} \quad (\textsc{streamDrop})$$

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa\ \texttt{Bool} \qquad op \in \{\texttt{not}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \kappa\ \texttt{Bool}} \quad (\textsc{streamOp1Bool})$$

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa\ \tau \qquad op \in \{\texttt{complement}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \kappa\ \tau} \quad (\textsc{streamOp1Bitwise})$$

$$\frac{\Gamma \vdash x : \texttt{Integral}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa\ \tau \qquad op \in \{\texttt{abs}, \texttt{signum}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \kappa\ \tau} \quad (\textsc{streamOp1Num})$$

$$\frac{\Gamma \vdash x : \texttt{Fractional}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa\ \tau \qquad op \in \{\texttt{recip}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \kappa\ \tau} \quad (\textsc{streamOp1Fractional})$$

$$\frac{\Gamma \vdash x : \texttt{Floating}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa\ \tau \qquad op \in \{\texttt{exp}, \ldots, \texttt{atanh}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \kappa\ \tau}$$
$$(\textsc{streamOp1Floating})$$

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa\ \tau_1 \qquad \tau_2 \in \Psi(\tau_1)}{\Gamma \vdash \texttt{cast}_{\tau_2}\ x : \texttt{Stream}\ \kappa\ \tau_2} \quad (\textsc{streamOp1Cast})$$

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa\ \tau_1 \qquad \tau_2 \in \Phi(\tau_1)}{\Gamma \vdash \texttt{unsafeCast}_{\tau_2}\ x : \texttt{Stream}\ \kappa\ \tau_2} \quad (\textsc{streamOp1UnsafeCast})$$

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa_1\ \texttt{Bool} \qquad \Gamma \vdash y : \texttt{Stream}\ \kappa_2\ \texttt{Bool} \qquad op \in \{\texttt{||}, \texttt{\&\&}, \texttt{`xor`}, \texttt{==>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \texttt{Bool}}$$
$$(\textsc{streamOp2Bool})$$

$$\frac{\Gamma \vdash x : \texttt{Integral}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Integral}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{`mod`}, \texttt{`div`}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$
$$(\textsc{streamOp2Integral})$$

$$\frac{\Gamma \vdash x : \texttt{Fractional}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Fractional}\ \ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{/}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$
$$(\textsc{streamOp2Fractional})$$

$$\frac{\Gamma \vdash x : \texttt{Floating}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Floating}\ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{**}, \texttt{`logBase`}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$

(streamOp2Floating)

$$\frac{\Gamma \vdash x : \texttt{Num}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Num}\ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$

(streamOp2Num)

$$\frac{\Gamma \vdash x : \texttt{Eq}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Eq}\ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{==}, \texttt{/=}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \texttt{Bool}}$$

(streamOp2Eq)

$$\frac{\Gamma \vdash x : \texttt{Ord}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Ord}\ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \texttt{Bool}}$$

(streamOp2Ord)

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \kappa_2\ \tau \qquad op \in \{\texttt{.\&.}, \texttt{.|.}, \texttt{.\^{}.}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$

(streamOp2Bitwise)

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \kappa_1\ \tau \qquad \Gamma \vdash y : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \kappa_2\ \tau_1 \qquad op \in \{\texttt{.>>.}, \texttt{.<<.}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$

(streamOp2BwShift)

$$\frac{\texttt{k}\ \text{is a}\ \langle\textsc{VInt}\rangle\ \text{token} \qquad \texttt{k}\ \geqslant\ 0 \qquad \Gamma \vdash x : \texttt{Stream}\ \kappa_1\ \texttt{(Array k}\ \tau\texttt{)} \qquad \Gamma \vdash y : \texttt{Stream}\ \kappa_2\ \texttt{Word16}}{\Gamma \vdash x\ .!!\ y : \texttt{Stream}\ (min\ \kappa_1\ \kappa_2)\ \tau}$$

(streamOp2Array)

$$\frac{\Gamma \vdash x : \texttt{Stream}\ \kappa_1\ \texttt{Bool} \qquad \Gamma \vdash y : \texttt{Stream}\ \kappa_2\ \tau \qquad \Gamma \vdash z : \texttt{Stream}\ \kappa_3\ \tau \qquad op \in \{\texttt{mux}\}}{\Gamma \vdash op\ x\ y\ z : \texttt{Stream}\ (min\ (min\ \kappa_1\ \kappa_2)\ \kappa_3)\ \tau}$$

(streamOp3)

$$\frac{\texttt{s}\ \text{is a}\ \langle\textsc{String}\rangle\ \text{token} \qquad \Gamma \vdash x : \texttt{Stream}\ \kappa_1\ \tau_1 \qquad \Gamma \vdash f : \tau_1\ \texttt{->}\ \texttt{Field s}\ \tau_2}{\Gamma \vdash x\#f : \texttt{Stream}\ \kappa_1\ \tau_2}$$

(streamStructAccess)

$$\frac{}{\Gamma \vdash \texttt{Nothing} : \texttt{Maybe}\ \tau}$$

(sampleVNothing)

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{Just}\ v : \texttt{Maybe}\ \tau}$$

(sampleVJust)

$$\frac{\Gamma \vdash x_1, ..., x_n : \tau}{\Gamma \vdash \texttt{[}x_1\texttt{,}...\texttt{,}x_n\texttt{]} : \texttt{[}\tau\texttt{]}_n} \qquad (\textsc{valueList})$$

$$\frac{x \text{ is a } \langle \text{VB}\textsc{ool} \rangle \text{ token}}{\Gamma \vdash x : \texttt{Bool}} \qquad (\textsc{valueBool})$$

$$\frac{i \text{ is a } \langle \text{VI}\textsc{nt} \rangle \text{ token}}{\Gamma \vdash i : \texttt{Integer}} \qquad (\textsc{valueInt})$$

$$\frac{x \text{ is a } \langle \text{VF}\textsc{loat} \rangle \text{ token}}{\Gamma \vdash x : \texttt{Float}} \qquad (\textsc{valueFloat})$$

$$\frac{\texttt{k} \text{ is a } \langle \text{VI}\textsc{nt} \rangle \text{ token} \qquad \texttt{k} \geqslant 0 \qquad \Gamma \vdash x_0 : \tau \ \ldots \ \Gamma \vdash x_k : \tau}{\Gamma \vdash \texttt{array} \ [\ x_0, \ldots, x_k \ ] : \texttt{Value} \ \ (\texttt{Array k } \tau)} \ (\textsc{valueArray})$$

$$\frac{\begin{array}{c}\texttt{k}_0, \ \ldots, \ \texttt{k}_n \text{ are } \langle \text{S}\textsc{tring} \rangle \text{ tokens} \qquad \Gamma \vdash x_0 : \texttt{Field k}_0 \ \tau_0 \ \ldots \ \Gamma \vdash x_n : \texttt{Field k}_n \ \tau_n \\ \Gamma \vdash \texttt{c} : \texttt{Field k}_0 \ \tau_0 \ \texttt{->} \ \ldots \ \texttt{->} \ \texttt{Field k}_n \ \tau_n \ \texttt{->} \ \tau\end{array}}{\Gamma \vdash c \ x_0 \ \ldots \ x_n : \tau}$$

(valueStruct)

$$\frac{\texttt{k} \text{ is a } \langle \text{S}\textsc{tring} \rangle \text{ tokens} \qquad \Gamma \vdash x : \tau}{\Gamma \vdash \texttt{(Field } x\texttt{)} : \texttt{Field k } \tau} \qquad (\textsc{field})$$

$$\frac{s \text{ is a } \langle \text{S}\textsc{tring} \rangle \text{ token}}{\Gamma \vdash s : \texttt{String}} \qquad (\textsc{string})$$

B.2 Haskell Typing Rules

The above typing rules help us understand when a stream in Copilot is ill-formed. The types implemented in Copilot as a Domain Specific Language embedded in Haskell do not include the length or lookahead as part of the stream. While the translation from one to the other is trivial, we include it here for completeness. For every typing rule, the type above the line is a dependent type, and the type below the line is a Haskell type. We assume that typing rules are listed by order of precedence. In particular, the last typing rule only applies if no other typing rule can apply, and any type that is not a dependent type in the previous rules is a valid type in Haskell.

$$\frac{\Gamma \vdash s : \texttt{Stream } \kappa \ \tau}{\Gamma \vdash s : \texttt{Stream } \tau} \qquad (\textsc{stream})$$

$$\frac{\Gamma \vdash s : [\ \tau\ ]_{\kappa}}{\Gamma \vdash s : [\ \tau\ ]} \qquad (\textsc{list})$$

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash s : \tau} \qquad (\textsc{any})$$

B.3 Safe and Unsafe Casting

From	To
Bool	Bool, Int16, Int32, Int64, Int8, Word8 , Word16, Word32, Word64
Int8	Int8, Int16, Int32, Int64
Int16	Int16, Int32, Int64
Int32	Int32, Int64
Int64	Int64
Word8	Int16, Int32, Int64, Word8, Word16, Word32, Word64
Word16	Int32, Int64, Word16, Word32, Word64
Word32	Int64, Word32, Word64
Word64	Word64

Table B1: List of safe castings in the Copilot language (Ψ).

From	To
Int8	Double, Float, Word8
Int16	Double, Float, Int8, Word16
Int32	Double, Float, Int16, Int8, Word32
Int64	Double, Float, Int16, Int32, Int8, Word64
Word8	Double, Float, Int8
Word16	Double, Float, Int16, Word8
Word32	Double, Float, Int32, Word16, Word8
Word64	Double, Float, Int64, Word16, Word32, Word8

Table B2: List of unsafe castings in the Copilot language (Φ).

Appendix C

Denotational Semantics

This section provides a denotational semantics for the key constructs in Copilot. We restrict ourselves to Copilot as described in the grammar provided earlier, without including all possible constructs available in Haskell.

The semantic domain of Copilot specifications is that of functions from natural numbers to values, where the natural number denotes the number of the sample in the stream (informally, the "time"). We therefore give meaning to Copilot's syntactic expressions by defining the function:

$$[\![\cdot]\!] :: \texttt{Stream}\ \alpha \to (\mathbb{N} \to \alpha)$$

Triggers would map to the same domain, and specifications to lists of functions in this domain.

We overload the same function to assign meaning to other constructs in the language, like literals, and functions operating on Booleans and numbers, for which we assume the obvious meaning, normally as defined in Haskell. For example, the meaning of $[\![\mathit{not}]\!]$ is just the Boolean function `not :: Bool -> Bool`. We consider most of these functions uninteresting for the purposes of discussing the semantics of the language and therefore leave them out of this chapter.

We assume the existence of a function *externDict* that, for a given name, returns a function from $\mathbb{N}$ to values of the necessary type. This function is partial, and its meaning depends on the type of the values returned. A proper treatment of this aspect would require knowing the type of elements in the stream, and using a family of functions *externDict* indexed by that type. For the treatment of fields, we assume to have available a function $\mathit{unField} : \texttt{Field}\ s\ k \to k$ that simply unwraps the value in a field.

$$
\begin{array}{rcl}
[\![\texttt{(}exp\texttt{)}]\!] & = & \lambda t \rightarrow [\![exp]\!]\ (t) \\
[\![\texttt{constant}\ exp]\!] & = & \lambda t \rightarrow [\![exp]\!] \\
[\![\texttt{extern}\ name\ samples]\!] & = & \lambda t \rightarrow (externDict(name))\ (t) \\
[\![ls\ \texttt{++}\ exp]\!] & = & \lambda t \rightarrow \begin{cases} [\![ls]\!]\ (t) & \text{if } t < length(ls) \\ [\![exp]\!](t - length(ls)) & \text{otherwise} \end{cases} \\
[\![\texttt{drop}\ k\ exp]\!] & = & \lambda t \rightarrow [\![exp]\!](k + t) \\
[\![op1\ exp]\!] & = & \lambda t \rightarrow [\![op1]\!]([\![exp]\!]\ (t)) \\
[\![exp1\ op2\ exp2]\!] & = & \lambda t \rightarrow [\![op2]\!]([\![exp1]\!]\ (t), [\![exp2]\!]\ (t)) \\
[\![op3\ exp1\ exp2\ exp3]\!] & = & \lambda t \rightarrow [\![op3]\!]([\![exp1]\!]\ (t), [\![exp2]\!]\ (t), [\![exp3]\!]\ (t)) \\
[\![exp\ \#\ id]\!] & = & \lambda t \rightarrow unField([\![id]\!]([\![exp]\!]\ (t)))
\end{array}
$$

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-02-2020	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Copilot 3	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ivan Perez and Frank Dedden and Alwyn E. Goodloe	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L–21127
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM–2020–220587

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 64
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
Ultra-critical systems require high-level assurance, which cannot always be guaranteed in compile time. The use of runtime verification (RV) enables monitoring these systems in runtime, to detect property violations early and limit their potential consequences. The introduction of monitors in ultra-critical systems poses a challenge, as failures and delays in the RV subsystem could affect other subsystems and threaten the mission as a whole. This paper presents Copilot 3, a runtime verification framework for real-time embedded systems. Copilot monitors are written in a compositional, stream-based language with support for a variety of Temporal Logics (TL), which results in robust, high-level specifications that are easier to understand than their traditional counterparts. The framework translates monitor specifications into C code with static memory requirements, which can be compiled to run on embedded hardware. This paper presents version 3 of the Copilot language, demonstrates its suitability with a number of examples, and discusses its use in larger applications. Additionally, it describes the framework's architecture, its implementation as a Domain Specific Language (DSL) embedded in Haskell, and the progress of the project over the years.

15. SUBJECT TERMS
runtime verification, formal methods, embedded systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Information Desk (help@sti.nasa.gov)
U	U	U	UU	41	**19b. TELEPHONE NUMBER** ***(Include area code)*** (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18