Compiling an Haskell EDSL to C

A new C back-end for the Copilot runtime verification framework

Frank Dedden[1,3]
Student number: 3705269

Mentor:
Alwyn Goodloe[2]

Supervisor:
Wouter Swierstra[1]

Second examiner:
Jurriaan Hage[1]

May 18, 2018

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

[1] Utrecht University
Department of Information and Computing Sciences
Utrecht, The Netherlands

[2] NASA Langley Research Center
Safety Critical Avionics Systems Branch
Hampton, Virginia, USA

[3] National Institute of Aerospace
Hampton, Virginia, USA

Abstract

Electronics and computers are all around us, ranging from cellphones to aircraft avionics. Some systems, including aircraft and nuclear powerplants, are considered safety-critical: failure can lead to the injury or death of humans. These ultra-critical systems require a very small probability of failure ($< 10^{-9}$), and are strictly certified by regulatory bodies.

Static software verification can aid in improving the safety of these systems, but only scales for a limited number of properties. Functional verification often requires the use of interactive theorem provers and are not practical for industrial scale systems. There is also a push to apply highly nondeterministic techniques such as machine learning in safety-critical systems, which cannot be verified using traditional proof techniques. Runtime verification, where specifications are checked during execution can allow us to verify software systems that cannot be verified by other means. The Haskell based Copilot runtime verification framework has been developed as part of a NASA project applying runtime verification to real-time embedded C programs. A lack of first class support for arrays and structs limited Copilot's practical usability. By replacing the current Atom and SBV-based code generation back-ends by a new, custom designed one, we are able to implement both arrays and structs as first class members of our specification. Both C structs and arrays are not straightforward to represent in Haskell, and require special attention. In particular type literals have been used to implement a safer array than Haskell normally allows.

Having native support for structs and arrays simplifies monitor specifications, and makes writing them easier and quicker. In addition, a tailored code generator also allows us to improve on the readability of the generated output code, making debugging and tracing easier, which in the end aids in making systems safer.

Contents

1 Background

In this section we provide a general background information for the topics discussed in this thesis. Section 1.1 surveys safety-critical embedded systems, which are the domain targeted for Copilot. Section 1.2 briefly introduces basic concepts from formal methods. Section 1.3 provides a short history of the field of runtime verification.

1.1 Safety-Critical Embedded Systems

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*, that is where failure can result in injury or death of a human, warrant special attention [Knight, 2002]. Society judges some safety-critical systems serious enough to subject them to government regulation and oversight. Systems that require a very small probability of failure (lower than 10^{-9}), such as civil transport aircraft and nuclear power stations are called *ultra-critical*, while ones that actually meet this requirement are called *ultra-reliable*.

Ultra-reliable systems often must be certified by a regulatory body as meeting the required levels of safety. In the case of civil transport aircraft, there is a set of standardized software design and development guidelines that, if followed, can help ensure the system to be certified. For instance the guidance document SAE4761 [SAE, 1996] designates a number of safety analyses that must be performed such as a hazard analysis [Leveson, 2012] and DO-178 [RTCA, 2011] describes many processes that need to be followed in the software development and testing processes. The grave consequences of failure have compelled industry and regulatory authorities to adopt conservative design approaches and exhaustive verification and validation (V&V) procedures to prevent mishaps. A particular stress in DO-178 is traceability of code to requirements and testability. Getting the requirements correct for large safety-critical systems is a very difficult engineering task in and of itself and ensuring that the software has a probability of failure (lower than 10^{-9}) is as much an art as a science.

In theory, software can be made "perfect", but in the case of embedded systems there are many assumptions about the operating environment that can be violated once in use and both the computer hardware and the system it is embedded in are subject to wear and tear as well as failure. Ultra-reliable systems must continue to function safely under such conditions. The terms 'failure', 'error', and 'fault' have technical meanings in the fault-tolerance literature. A *failure* occurs when a system is unable to provide its required functions. An *error* is "that part of the system state which is *liable to lead to subsequent failure*," while a *fault* is "the *adjudged or hypothesized cause* of an error" [Laprie, 1995]. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to system failure. A *fault-tolerant system* [Butler, 2008] is one that continues to provide its required functionality in the presence of faults (for the faults tolerated). A fault-tolerant system must not contain a *single point of failure* such that if the single subsystem fails, the entire system fails. Thus, fault-tolerant systems are often implemented as distributed collections of nodes such that a fault that affects one node or channel will not adversely affect the whole system's functionality. Ultra-reliable systems achieve their high-level of reliability through a combination of hardware redundancy and very sophisticated algorithms. Unfortunately, this complexity of the solution means it is often more difficult to implement than the actual system.

1.2 Formal Methods

Formal methods of program specification and verification has its origins in the following two observations: first, that testing can show the presence of bugs, but cannot demonstrate their absence, and second, if programs and their specifications are expressed in a logical, formal language, then program correctness can be demonstrated using formal mathematical proof [Floyd, 1967, Hoare, 1969]. Great progress has been made in developing a range of mathematical techniques for ensuring varying levels of correctness of both hardware and software and these techniques are gaining increased acceptance in industry especially in safety-critical domains such as aerospace. For instance the DO-333 [RTCA, 2011] supplement to DO-178 provides guidance on how formal methods can be used instead of testing for a significant percentage of the testing required to obtain certification for civil aircraft.

Static analysis tools such as abstract interpretation [Cousot and Cousot, 1977] can identify a significant class of bugs in software, but current techniques tend to yield many false positives. Model checking [Clarke et al., 1999], where a design is captured as a state machine and sophisticated algorithms are employed to check that this model satisfies a temporal logic specification, has been especially effective in verifying safety properties of computer hardware and protocols. In addition, model checking has been successfully used in may other application domains, but model checking can be very sensitive to state explosion. The aforementioned fully automatic tools are often not suitable for proving the functional correctness of a system. For instance, this often requires the use of interactive theorem provers such as Coq [Bertot and Castran, 2010] and PVS [Owre et al., 1992], that require a lot of effort to for a programmer to become proficient with. However, there is a growing body of experts in the area who have had many notable successes ranging from verified compilers to verified industrial floating-point units to verified airspace management algorithms.

1.3 Runtime Verification

In spite of many notable successes in applying formal methods to industrial systems, there are practical limitations. Formal methods are often the most effective when applied to requirements, but given limited resources, the code itself is not subject to formal proof. When applying formal methods to the design many assumptions are made, these must often be checked at run time. In the case of very large systems, it is not yet practical to apply formal methods to the whole system. Advances in artificial intelligence are enabling the development of increasingly autonomous cyber-physical systems that modify their behavior in response to the external environment and learn from their experience. While unmanned aircraft systems (UAS) and self-driving cars have the potential of transforming society in many beneficial ways, they also pose new dangers to public safety. The algorithmic methods such as machine learning that enable autonomy lack the salient feature of predictability since the system's behavior depends on what it has learned.

Runtime verification (RV) [Goodloe and Pike, 2010], 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. Runtime verification ensures that properties are not violated at runtime so it cannot be viewed as a proof of correctness, but a significant improvement over testing alone. The *Simplex Architecture* [Sha, 2001] provides an architecture design 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.

2 Introduction

Copilot [Pike et al., 2010, Pike et al., 2011, Pike et al., 2013] is a runtime verification framework that was developed as part of a research project investigating and applying runtime verification to safety-critical hard real-time embedded systems. The project was initiated in 2010 under NASA research by principal investigators Dr. Lee Pike (Galois Inc.) and Dr. Alwyn Goodloe (National Institute of Aerospace/NASA). The project consists of a collection of Haskell libraries that provide an embedded domain specific language (EDSL) for defining monitors for hard real-time embedded systems. The Copilot framework translates high-level specifications of system safety into monitors implemented in a subset of C that runs in constant time and constant space on real-time embedded systems. To date it has been applied to monitor software on an avionics testbed and on various flights of an unmanned aircraft system (UAS).

While Copilot started out solely as a research project, over the years engineers found more and more interest in the project as well. With the increased use outside of research, several limitations of the current implementation arose. The bigger deficiencies, no support for structs and arrays and in some cases huge inefficiencies take a lot more effort to fix. The solution to these problems is clear: rewrite the compiler of Copilot, so we can have proper support for structs and arrays, and improve on efficiency of the produced code as well.

If runtime verification is to be used when other forms of verification are either impractical or impossible, it is very important for us to be able to trust the monitors we have written. Hence we want to be able to formally verify that the generated monitors indeed implement the specification. This can be very difficult to do by hand, consequently we would like to employ a verification tool for this.

The aim of this project is to find a solution to these problems, while improving on the implementation in general. This will open up new possibilities for the application of Copilot.

2.1 Research question

The most important part of this project will be reimplementing the code generator. Due to the nature of the project, we will not be focusing on a single specific research question. We will define a broad question that covers the project, but is subdivided in smaller research questions that each focus on the sub problems. We define our main research question as follows: *How to translate Haskell monitoring code for embedded systems to C*? We can divide the problem into these questions:

- Non-scalar values, like structs and arrays, can not be directly translated from Haskell to C; how do we support these?
- How do we use a formal verification tool to verify the generated code?
- Can we improve on the traceability of the code, without having the user of Copilot do extra work?

3 Copilot

This section provides a brief introduction to the Copilot runtime verification framework intended to communicate the core ideas needed to understand the later sections of this thesis.

3.1 Overview

Copilot consists of a collection of Haskell libraries that provide an embedded domain specific language (EDSL) for defining monitors for hard real-time embedded system. The defined monitors do not monitor hardware directly but are interfaced with C variables and functions. This approach has two main benefits: first, there is no need to implement hardware specific features, and second, our monitor is not limited to sensors, so we can monitor every possible C variable. Copilot can be seen as a generalization of the idea of Lustre's [Caspi et al., 1987] "synchronous observers" [Halbwachs et al., 1993], which are Boolean-valued streams used to track properties about Lustre programs. Whereas Lustre uses synchronous observers to monitor Lustre programs, Copilot applies the idea to monitoring arbitrary periodically-scheduled real-time systems.

Most RV frameworks heavily instrument code so each update to a monitored variable is sent to the monitor. This is not feasible in avionics and similar domains. Instead, Copilot samples variables. 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 [Dwyer et al., 2008]. 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. Real-time programs usually 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, false positives are possible, while 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. In practice, this approach has worked very well for Copilot since the engineers who use it are experienced at hard real-time scheduling.

In order to interface with the C code, we need to translate our Haskell specification to C, so it can be included from the C code base. The translation from the front-end Copilot monitor, through underlying representations to the final C code is done in several steps. Figure 1 shows an overview of the modules in Copilot.

Core At the heart of the Copilot EDSL is Copilot *Core*, this contains a low level representation of the specification. This specification is translated to C code through one of the two back-ends, but for testing purposes it can also be run in an interpreter.

Libraries/Language The front-end of Copilot is defined in *Language*. It is a set of functions that makes working with the specification of Core a lot easier. Users of Copilot will always use these functions to write their specification, instead of speaking to Core directly. Alongside Language, Copilot contains a number of more high level libraries built on top of Language.

Atom back-end This code generating back-end is built upon the Atom library [Hawkins, 2008] by Tom Hawkins, used for generating hard real-time C code from Haskell. Currently this back-end is unmaintained, and although it produces code that works, the output is very hard to read.

SBV back-end This back-end generates code using Haskell's SBV library, by Levent Erkök. SBV is a library to express properties and prove them with an SMT solver. Currently this back-end is favored over the Atom one, as it is being maintained, and it can be used to express properties on the specification and have them proved by a solver like Z3.

Interpreter The interpreter allows us to simulate the execution of a monitor, without actually translating it to C. During testing this allows us to run the monitor on a regular machine, instead of an embedded one. The output of it can be tested and compared with the back-ends using the QuickCheck [Claessen and Hughes, 2000] framework.

Pretty Printer The pretty printer displays a slightly rewritten and more humanly readable version of a Core specification. This could aid in debugging monitors.

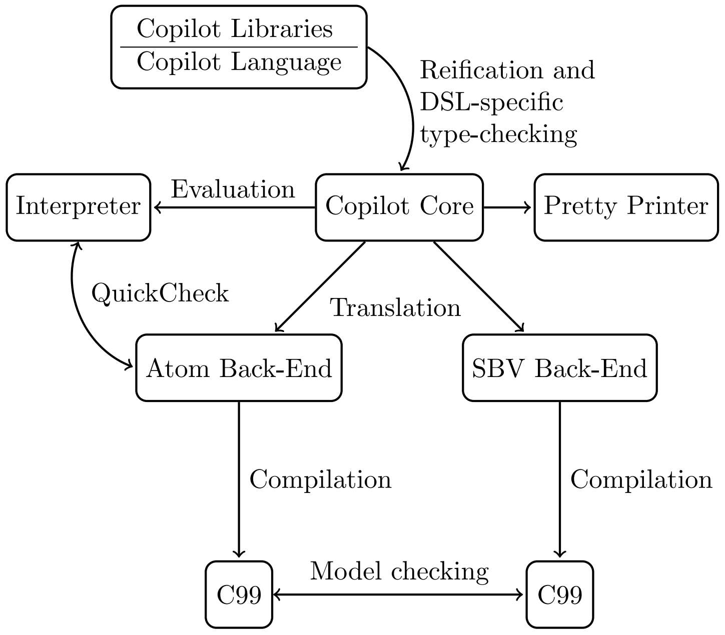

Figure 1: The Copilot toolchain

3.2 Specifications

A Copilot specification is a program written in a reactive EDSL. Functional reactive programming is a form a declarative programming where we write programs based around data streams. Explicitly modelling time allows us to define streams depending on time. The Copilot EDSL is based around streams as well, but differs slightly in the sense that it does not explicitly model time.

In Copilot, the specification we write defines which external variables from C are used, and how these are combined and modified in case calculations need to be made. These variables are represented using streams, which are very similar to infinite lazy lists. In Copilot a single step, or iteration, of a stream models a single step forward in time. As Copilot is designed in such a way that the duration of a single iteration is always constant, the streams determine our monitoring frequency.

Streams can be combined and modified using a number of operators. Most of these operators are analogous to the ones defined for numerals and lists. In addition, literal values are lifted to streams automatically, this allows us the write programs on streams with relatively little syntactic overhead:

```
x :: Stream Int32
x = 1
```

This will yield the constant stream containing only ones. In similar ways, the basic numerical operators like `+`, `*` etcetera are overloaded, and can be applied directly to streams. The operator `++` is used to append a list and a stream. Using these operators we can define the Fibonacci sequence without much effort. Note that this function calculates the sequence over time, each stream iteration the next element of the sequence is calculated:

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

The definition of `fib` might seem surprising at first, but after a closer look it is quite simple. We start of with defining a buffer of `[1, 1]`, which are the first two elements of our sequence. We append a stream, defined by two recursive calls, to this buffer. We drop the first element from the second call, effectively skipping the first element and jumping forward in time by one step. The result is added to the first recursive call:

$$\begin{array}{rcccccccccc}
\texttt{fib} & \langle & 1, & 1, & 2, & 3, & 5, & 8, & 13, & \ldots & \rangle \\
\texttt{drop 1 fib} & \langle & 1, & 2, & 3, & 5, & 8, & 13, & 21, & \ldots & \rangle \\
\texttt{fib + drop 1 fib} & \langle & 2, & 3, & 5, & 8, & 13, & 21, & 34, & \ldots & \rangle
\end{array}$$

Figure 2: Calculating the Fibonnaci sequence

One could imagine that we can drop any amount of elements from a stream, but this is not the case. Future elements from a stream may depend on external values, and by dropping elements from externals streams we need to be able to look in the future, which for obvious reasons is not possible. In practice we can drop elements from a stream as long as the buffer always contains at least one element. In other words: the amount n we drop must always be positive, but smaller then the length of the buffer. `fib` has a buffer of two elements, therefore we can only drop a single element maximum.

Modelling external variables as streams is done using the `extern` function. This function takes a name of a C variable and a possible list of default values. It comes in several flavours, each of them for loading specific types of data. At some point we need to do something with the data we read. Using triggers we can call previously defined C functions, based on the value of a stream. The resulting type of `trigger` is `Language.Spec`, which is a monad provided by Copilot used to keep track of triggers. In order to use the spec, we have to `reify` it to the lower level `Core.Spec` representation first, which in turn gets compiled by the `compile` function, implemented in the chosen back-end.

The following example illustrates the usage of many of the Copilot capabilities combined in a single monitor specification for a heating element. Note that `temperature`, `heat_on` and `heat_off` are all defined in the C program that actually interfaces the heater. In this example, we rely on the `C99` back-end to generate our code. In order to use the produced code, we need to include the generated header file, call the `step()` function inside our main-loop and compile all the sources with a C compiler.

Listing 1: Heater example

```haskell
module Main where

import Language.Copilot
import Copilot.Compile.C99

import qualified Prelude as P

temp :: Stream Int32
temp = extern "temperature" Nothing

spec = do
  trigger "heat_on"  (temp < 19) []
  trigger "heat_off" (temp > 21) []

main = do reify spec >>= compile defaultParams
```

Copilot does support more constructs and operators, but these are not necessary to understand the remainder of the thesis. The tutorial on Copilot [Wegmann et al., 2015] provides more information on this matter.

3.3 Current limitations

The current implementation of Copilot started out as a project that has been extended over the years. New possibilities and features have been added by numerous people. Most projects that are developed in such a manner are in need of a code refactoring at some point. It is not possible to plan for every feature from the very start, nor can we require that every developer has the same programming style. Sadly, Copilot is no exception to this rule.

As described in section 3.1, Copilot has two back-ends available for the generation of C code: *C99* and *SBV*. Currently both back-ends support the same features, but only the SBV based one is maintained. The Atom based C99 back-end works, but has not been maintained for a couple of years. Both back-ends have a limited set of supported features: both only support scalar values, so no structs or arrays. This is a rather large problem, as most flight control systems in aircrafts rely on matrices, which are either implemented by arrays or structs. In practice this means that users of the library have to write functions to pack and unpack such data structures, on both the C and Haskell side. Obviously this is a lot of tedious and bug-prone work.

Another important shortcoming of the current back-ends is that they produce code that is not efficient. C99 being based on Atom, creates code that is constant in memory and time, and does its own scheduling. Because of this, code generated by Atom deviates quite a lot from the original definition. As the implementation of the C99 back-end is pretty straightforward, no optimizations are done on the specification. In combination with the way Atom is designed, we end up with code that is not optimized for our purpose. The SBV back-end creates more readable code, but everything is scattered over a large number of files, making it hard to get a overview of the code.

Connected to the readability of the code is traceability: we want to be able to tell which part of the produced code originates from which part of the specification. Having readable code helps us understand the program batter, and it is easier to see how parts relate to the original specification. Traceability helps not only while debugging the generated code, but also gives us a hint which part of the specification may take more time and memory than other parts. Both of the current back-ends do not support good traceability. As explained, C99 is based on Atom, which is its own EDSL for defining programs. The generated code differs much from the original specification, and because of that generated variable names are used. This makes finding the original over variables and other pieces of code so much harder. The SBV back-end however produces nicer code, but as explained before, due to the scattered nature, it is very hard to understand.

We want to be able to verify our specification. Currently the SBV back-end can be verified using SBV. SBV does not support structs and arrays however, so we cannot rely on SBV for this purpose. The C99 back-end does not support any verification language at all. Instead we want to incorporate generation of ANSI/ISO C Specification Language (ACSL) [Baudin et al., 2015] rules within our C generation. Using the Frama-C verification tool [Frama-C webpage, 2018], we can than verify our monitoring specification.

4 Implementation

In this section, we discuss design options considered for implementing structs and arrays in Copilot. For both structs and arrays we will introduce requirements criteria that had to be met in the implementation. Several implementation techniques are presented in some depth and engineering trade-offs are discussed in order that the reader may better understand why we made the design decisions we made.

4.1 Structs

Structs are an important feature of the C programming language, and are used to pack multiple values into a single variable. In avionics this is useful to represent vectors, matrices and coordinates for example. If we want to monitor structs with Copilot, we need to be able to represent them in Haskell. Let us first take a look what defines structs:

1. A container that packs different fields together.
2. Each field has its own name and type, where the type can be any valid type. These can be structs and arrays themselves as well.
3. We have to access each field, allowing us to read and modify them.
4. For code generation purposes, we need to be able to have the field names available as strings in Haskell.
5. They are represented by a unique type, distinct from any other types of structs.

Conditions 1 through 4 should be self explanatory, as they mimic C structs, the fifth condition however is specific to Copilot. If we want to be able to take a specific field of the struct, we need to translate our Haskell definition to the equivalent C code. In order to do so, we need to know the name of the field:

```
int a = struct.field;
```

There are roughly two ways to model structs in Copilot, both having their advantages and disadvantages:

1. Use a single stream for a struct, modeling the struct as an indexed list.
2. Rely on Haskell datatypes to model a struct, thus ending up with a stream of these datatypes.

We will discuss these approaches in the next sections.

4.1.1 Indexed lists

Let us first take a look at the first approach, where we model structs by simply using an indexed list of values. The name of the field is used as the index, while both the value and type of the field are actually stored as values in the list. We introduce the `Type` datatype, which we need to represent the type of the field. `Type` therefore has a direct relation with the types known by C. While technically we could have used `Bool` and `Int8` as constructor names of `Type`, here we prepend the names with a `T`, to make a clear distinction with the regular Haskell types:

```
type VarName = String
data Type    = TBool | TInt8 | TInt16 | ...

data Value  = forall a. V Type a
type Struct = [(VarName, Value)]

v :: Struct
v = extern "v" Nothing
```

In order to create a list of different types of values (a heterogeneous list), we need to hide the actual type (e.g. a Haskell-level `Int` or `Double`) behind a constructor. In this case we use `Value` to combine a literal Haskell value with its C type. By introducing an existential quantification on the value of type `a`, we can omit it as a parameter to `Value`. As the type of `Value` does not contain any information on the type that it contains, we can use it to type heterogeneous lists: `[Value]`. Additionally, we need to enable GHC's `ExistentialQuantification` extension to be able to use the `forall` keyword.

Judging by the requirements for structs we defined earlier, indexed lists seem like a viable solution to implementing structs in Copilot:

1. It is possible to pack multiple fields together in one container.
2. By hiding the actual type using a quantified datatype like `Value`, we can store values of different types. The `Type` datatype is used to store the type, so we can pattern match on them if necessary.
3. Storing the elements in a list allows us to lookup specific fields by name, and modify them.
4. The names of the fields are available as strings in Haskell, which is necessary for generating C code that reads or updates these specific fields.

Unfortunately though, the approach does not meet the fifth requirement, as it uses a generic `Struct` type. Not only do all structs share this type, the list does not provide us with much type-safety either. We can add and remove items as we wish:

```
-- A 2D vector
v2 :: Struct
v2 = [("x", V TFloat 1.0), ("y", V TFloat 2.0)]

v2' :: Struct
v2' = ("condition", V TBool False) : v2
```

While this may seem handy in some cases, this completely removes the idea of using Haskell and its type-safety to write monitors.

4.1.2 Structs using datatypes

We need to find a way to model structs, that provides more type-safety than lists, without violating any of the other requirements. Regular Haskell datatypes, preferably using record-syntax, allow us to model structs really well in Haskell:

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

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

Haskell's type system now forces correct use of struct types: we cannot add or remove fields to `Vec`, nor can we use it instead of any struct due to its distinct type. The strict types come with a downside as well: we are unable to write functions that are polymorphic on structs. When writing monitors this way will most likely not be a problem: monitors need to work on specific structs, i.e. we cannot take a value of an arbitrary struct. Internally in Copilot though, this limitation will lead to trouble. For example, how do we implement the `extern :: Typed a => String -> Maybe [a] -> Stream a` function, that reads an external struct? The solution is easy enough, we create a `Struct` class of which all structs need to be an instance:

Listing 2: Definition of the `Typed` class

```
1  class Typed a => Struct a where
2    typename      :: a -> Typename a
3    toValues      :: a -> Values a
4
5  data Typename a = TyTypedef String
6                  | TyStruct  String
7
8  type Values a = [Value]
9  data Value    = forall a. V (Type a) String a
```

Line 1 Our instance of `Struct` needs to be an instance of `Typed` as well. This class is used internally in Copilot, where streams can only hold data of a type that is an instance of `Typed`. Because we want to create a stream of our custom struct type, it needs to be an instance of `Typed`.

Line 2 For code-generation purposes, we need to be able to access the name of the struct, as used in C. This does not necessarily have to be equal to the name of the Haskell datatype, but that would be the most convenient. Note that `typename` is a function taking a value of type `a`, and returning a parametrised `Typename a`. Sometimes GHC is not able to infer the correct type of the parameter to `Typename`, for example in cases where we immediately translate the type name to something else like a string. Using a function for the type name allows us to pass an explicit value, providing GHC with enough informatin to find the correct instance.

Line 3 The names of the fields of the structs only exist at Haskell compile-time, but we need to access them at runtime in Haskell. The solution is to bind a string to each field, containing the corresponding C name. Instead of carrying the names of the fields around inside the struct data (similar to the approach of `Value` in the indexed lists), we opted to use a wrapper function: `toValues`.

Lines 5-6 In C a type of a struct can be written in two ways: by using the `struct` keyword, or using a typedef. Both need to be supported in Copilot, as both could be used by the C program we are monitoring. The `Typename` datatype allows us to use either of the two, and the output code will be written accordingly. It depends on the program we are monitoring how the structs are defined, therefore it is up to the programmer of the monitor to match this with the program.

By making `Typename` a phantom type, we make sure that the resulting type is parametrised with the type of the struct it belongs to. This makes it impossible to share a type name between different types of structs, increasing type-safety a little bit.

Lines 8-10 The `toValues` function produces a list to describe the fields of a struct. Because this list contains values of multiple types, we need to create a heterogeneous list. `Values` is a phantom type parametrised with the type of the struct that defines a list defining the types of the struct's fields.

`Value` is an interesting datatype in its own right in that it does not take a parameter, yet the arguments of its constructor are parametrised. Using the `forall` keyword, we create a so called *quantified type*, for which the argument is inferred by GHC. The purpose of `Value` is not only to hide the actual type of the value it holds, but also combine this with a term-level representation of that type. We do not want to lose this type information, so we can pattern match on it when needed. We could have used the same definition of `V` as we already did for our index list prototype. We did not choose to do this however, as it allows us to write values that do not introduce a type error, but is semantically illegal:

```
v :: Value
v = V TBool 10
```

The solution is to turn `Type` into a *generic algebraic data type* (GADT), where each constructor can have a distinct result type:

```
data Type :: * -> * where
  TBool  :: Type Bool
  TInt8  :: Type Int8
  TInt16 :: Type Int16
  ...

data Value = forall a. V (Type a) a
```

Because both arguments of `V` are bound by the same `forall`, we force them to be of the same type. With these modifications, one cannot write illegal values anymore:

```
v :: Value
v = V TInt8 True
```

```
* Couldn't match expected type 'TInt8' with actual
    type 'Bool'
* In the second argument of 'V', namely 'True'
  In the expression: V TInt8 True
  In an equation for 'v': v = V TInt8 True
```

Surprisingly, this parametrised version of `Type` is already available in Copilot, as it is used internally. In this report we prepend the constructors with a `T`, to make a clear distinction with the Haskell provided types.

Additionally, we need to extend the definition of `Type` with a constructor for structs. Note that the constructor takes an argument to specify the exact type of struct:

```
data Type :: * -> * where
  ...
  TStruct :: (Typed a, Struct a) => a -> Type a
```

Now that we have seen how the `Struct` type class is defined, let us take a look at a sample implementation of a simple vector datatype. This type is represented in C by the following definition:

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

And in Haskell using the following instances:

Listing 3: Example instances of `Struct` and `Typed`

```
1 instance Struct Vec where
2   typename _  = TyTypedef "vec"
3   toValues v  = [ V TFloat "x" (x v)
4                 , V TFloat "y" (y v)
5                 ]
6
7 instance Typed Vec where
8   typeOf = TStruct (Vec 0 0)
```

Line 2 We define the type name as a typedef of “vec” to match the definition in C.

Lines 3-5 The result of `toValues` is a list of the fields in the same order as defined in the data type.

Line 8 To complete the instance for `Typed`, we need to provide a definition to `typeOf`, which is used by the code generator. The interesting part is that we need to provide `TStruct` with something of the correct type. The easiest solution is to provide a dummy vector, for which the actual values inside the vector are ignored.

Now structs are real first-class in Copilot, so we treat them as such:

Listing 4: Example usage of structs

```
exvec :: Stream Vec
exvec = extern "exvec" Nothing

s :: Stream Vec
s = [Vec 1.0 2.0, Vec 3.0 4.0] ++ exvec
```

4.2 Arrays

Together with support for structs, support for arrays form the most important change in the new back-end. At first glance, arrays seem easier to implement then structs, being a collection of homogeneous values. In practice, arrays turn out to be harder to implement, mostly because we cannot define its type as strictly as we can for structs. In addition, we need to find a suitable representation for nested arrays like matrices. The problem of implementing arrays in Copilot is twofold: first off we need to find a suitable way to model arrays in Haskell. In addition we have to write C code that semantically behaves similar to the implementation for scalar values and arrays. We will see that this code is more complicated however, as arrays are not first class members of the C language.

Haskell does provide an array type, but not one that has a very precise relation with C arrays. Let us therefore define some requirements, so we can model C arrays as close as possible:

1. The array should be indexed only by unsigned integers, just like in C.
2. We would like to disallow nested arrays (i.e. an array of type `Array (Array a)`), but rather support real multi-dimensional arrays. Nested arrays have the downside, that it is impossible to force the sub arrays to be of equal length. The type does not force so called *rectangular arrays*. Just like with lists we can define non-rectangular arrays: `[ [1, 2], [4, 5, 6] ]`. While this is possible in C, it is not trivial and not really useful either.

   Multi-dimensional arrays do force arrays to be rectangular, as we use an index with multiple dimensions. These dimensions always specify a rectangular range of numbers, and therefore a rectangular array as well.

   We could force rectangular arrays using dependent types, by specifying the length of the array with its type. The inner arrays than would have a fixed length as well, i.e. `Array 2 (Array 4 a)`. Without relying on language extensions, Haskell does not (yet) allow type literals and dependent types, so we cannot force nested arrays to be rectangular. We could use these extensions, but as we will see later on, this will not completely fix our problems.

   Another problem with nested arrays, is that it makes generating C code harder. While C supports nested arrays, usage of deeply nested arrays requires a lot of pointer operations, obfuscating our code. The solution would be to flatten the arrays, but this introduces another problem: it is not possible to flatten an array using a regular function. This is similar to the problem of flattening an arbitrary list, it is not possible to come up with a correct type for such a function. Our best shot would be to rely on type classes, for which we write two instances:

   (a) A base case, which functions as an identity. It does not flatten an array.

   (b) An inductive case, which removes exactly one level, similar to `concat`.

While this approach works, it requires several language extensions as it relies on type classes with multiple parameters. In addition we end up with overlapping instances of the class, which is now possible due to the multi parameter classes. This has the downside that GHC is not able make a distinction between removing only a single nesting level, or flattening down to any other smaller level. To workaround is to force the result type of this expression, but this becomes very messy when this expressions is part of another one.

In the end this approach introduces to much trouble, making us decide to disallow nested arrays.

3. We prefer arrays to be of a fixed length, this makes it easier to translate these arrays to C, where they normally have a fixed length as well.
4. In addition, we prefer the length of the array to be part of its type. Later on we will see why is this useful for code generation.

With these requirements, we can now propose a number of possible solutions:

1. Use a dependently typed implementation to force correct lengths, as well as correct operations on the array.
2. Have a rather simple type that mimics the well known `Data.Array`, but is stricter on the values used as indices.
3. A solution that uses type-literals to store the length of the array.

4.2.1 Dependently typed arrays

Dependent types allow us to define types that have dependencies on both other types, as well as values. This is in contrast with a traditional type system, where we can only write types that are constructed from other types. Using dependent types, we could for example define types that force lists to be sorted, or natural numbers to be smaller than a specific value. While Haskell is not a dependently typed language, it supports a limited form of type dependency using a couple of GHC extensions.

Vectors or arrays are one of the first examples of programming in a dependently typed language. The idea is that we define an inductive type using a `Nil` and a cons case `(:>)`, just as with lists. The main difference is that we keep track of the length of the array by adding a natural number to the type of arrays. Our `Nil` case comes with a length of 0, while `(:>)` increases the length of its argument by one:

Listing 5: Implementation of dependently typed arrays

```haskell
data Array (n :: Nat) a where
  Nil   :: Array 0 a
  (:>)  :: a -> Array n a -> Array (n+1) a

infixr :>

arr :: Array 3 Int
arr = 1 :> 2 :> 3 :> Nil
```

Line 1 The definition of `Array` takes `a` as a type parameter, but also has an additional one, which is of kind `Nat`. In Haskell a kind, is the type of a type. In this case, each possible natural number is a constructor of the `Nat` kind.

Line 2 The `Nil` case constructs an empty array, which therefore has length 0.

Line 3 `(:>)` takes a single element of type `a`, an already existing array storing the same type, and creates an array which is exactly one element longer. Note that the we use the `+`-operator here, although similar in use as the one we already know, this takes natural numbers as its arguments, instead of instances of the `Num` class. Its kind therefore is `(+) :: Nat -> Nat -> Nat`.

Line 5 We define `:>` as an right associative infix operator, which allows us to chain applications to construct an array, like in line 8.

Lines 7 and 8 Define a example array of length 3. Note that this compiles correctly, as the data of our array is actually 3 elements long. If the length of the type and data do not match, GHC will present us with an error message that it cannot match the two types.

As we can see, dependent types allow us to write arrays in very safe manner. We do need to extend the definition with some helper functions, in order to be able to do something useful with the array. Let us try to define some auxiliary functions:

```
foldr :: (a -> b -> b) -> b -> Array n a -> b
foldr f b Nil       = b
foldr f b (x :> xs) = f x (foldr f b xs)
```

Indeed this implementation satisfies what we expect from `foldr`. It matches on the length of its argument. If it is empty, just return the base element, otherwise take the first element of the array and recurse on the rest. However, other basic functions will not be as easy to implement:

```
append :: Array n a -> Array m a -> Array (n + m) a
append Nil       ys = ys
append (x :> xs) ys = x :> append xs ys
```

```
* Could not deduce: ((m + n1) + 1) ~ (m + n)
  from the context: n ~ (n1 + 1)
```

One can easily see that given `n ~ (n1 + 1)`, `((m + n1) + 1) ~ (m + n)` is deducible, however GHC is not able to. Currently GHC, even with the correct extensions enabled, does not know about basic mathematical rules, and is not able to deduce it. In contrast to a type dependent language like Agda, GHC is not able to use other functions defined in the file as part of its type checker. For that reason, it is not possible to provide the knowledge GHC needs to deduce these cases, without relying on compiler plugins. We could get very specific cases to work though, if we would change the definition of the `+`-operator to either be left or right recursive. The sad truth is that this will conflict with definitions of other functions, like `take` or `drop`.

Christiaan Baaij has developed `ghc-typelits-natnormalise`, a plugin to the GHC type checker that can solve equalities between types, when these are either natural numbers, variables or arithmetic expressions. This plugin allows us to easily write functions like `append` in a type-dependent manner, however for the purpose of Copilot, we do not want to rely on external plugins and libraries too much. Copilot is a project of which people expect it to behave stable over several years, therefore we opted not to use this plugin.

4.2.2 Simple arrays

As we saw in the previous section, dependently typed arrays could be a really nice solution, if GHC actually had more support for it. Unable to use `ghc-typelits-natnormalise`, we are forced to find a solution that does not make use of these strict dependent types.

The most used array datatype for Haskell is provided by `Data.Array`. This datatype provides a fixed length array, with support for multi-dimensional indices. Its type takes two parameters, an index `i` and the type of elements `e`:

```haskell
data Array i e
array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

It does not export any constructors, forcing us to use the smart constructors that are exported by the library. The `array` constructor, takes a tuple which denotes the range of the index that is used. The `Ix` class is the class of types that can be used as an index. All basic Haskell datatypes are instances of this class, as well as tuples of those, up to a five-tuple. For a two dimensional array, e.g. a 4x4 matrix, this would be:

```haskell
mat :: Array (Int, Int) Float
mat = array ((0,3), (0,3)) [...]
```

Note that the array has a fixed length, defined by the range tuple, but this is not enforced by its type, but rather by the implementation of the smart constructor.

The `Data.Array` implementation is nice for regular Haskell programs, but is not really suited for use in Copilot. First of all, we want to restrict the possible indices to just natural numbers. This creates a stronger connection with the C language, which ensures that we can translate these indices to C. Second, we do not want to be able to specify a range of indices, as we want it to always start at 0.

Our own array implementation mimics the one from `Data.Array`:

Listing 6: Implementation of more limited arrays

```haskell
class Index i
instance Index Int
instance Index (Int, Int)

data Array i a where
  Array :: Index i => i -> [(i, a)] -> Array i a
```

We removed the range argument and replaced it with a single instance of the `Index` class. This argument specifies the dimension of the array. Judging by the instances of `Index`, only one and two dimensional arrays are currently supported, but more can be easily added.

The huge downside to this approach is that we never specify the actual size of the array. It is solely based on the size of our input data. This does not give us a lot of safety in terms of length, i.e. we could easily replace an array of the same dimension with one that has a different length.

Later on we even found a bigger problem: the length of the array is not encoded in its type. For external streams of arrays, we need to write C code that copies the elements of our external array into the local memory of the monitor. For this operation we need to know the length of the array, so that we know how many elements we need to copy (see section 5.4). External arrays do not have a value in Copilot, therefore it is not possible to know the length of an external array, unless it is either provided by the user at term level, or it is encoded in its type.

Providing the length at term level required making a lot of low level changes to Copilot, actually adding a whole new type of stream. This would not only be a lot of work, but an ugly solution as well. In addition, we would still have the problem that the length of arrays is not fixed, making our effort all in vain anyway. Therefore the only viable solution is to add the length to the type of the array.

4.2.3 Arrays using type literals

As we saw in the previous section, in the case of Copilot, there is a need for storing the length of the array within its type. We do prefer to rely on fully dependent types, but GHC's type checker is not up to the task yet. In the end we came up with a solution that is a combination of both approaches: we extend the arrays described in the previous section with type literals.

Let us take the simple definition of arrays from the previous section:

```
data Array i a = Index i => Array i a
```

We will keep using `Index` to denote the type of indices used, but we need to extend it with a type-literal indicating its length. This does not come without its problems, though:

1. The smart constructor still takes a list of tuples, combining an index with a value. We need to check if these indices correspond to the one given by the type.
2. We need to force the dimension specified in the type to be equal to the ones specified in the input list; that is, if our type specifies two dimensions, so must our input list.
3. We need to be able to retrieve the size of an array at value level, even when we are only given a type.

Points 1 and 2 pose an interesting problem: how do we create a relation between type literals and values? To understand that better, let us take a look at types and kinds. Let us try to define a naive implementation for the `Int` type in Haskell. The `Int` does have boundaries defined by the specific Haskell implementation that is used, but for now we will ignore those. We can interpret each and every integer number to be a constructor of this type. Using an algebraic datatype we come up with the following definition in pseudo-Haskell:

```haskell
data Int = ... | -2 | -1 | 0 | 1 | 2 | ...
```

Here the literal numbers are values that construct data of type `Int`. For a kind however, its constructors are not values, but types itself. Our problem is, is that there is no connection between `Int` and `Nat`. We are unable to create data with a specific type level natural number. This is no surprise: in Haskell types are defined at compile-time, and thus by definition cannot depend on runtime values. However we can request the current value based on a type:

```haskell
import GHC.TypeLits (natVal)
import Data.Proxy    (Proxy (..))

{- data Proxy a = Proxy -}
{- natVal :: KnownNat n => proxy n -> Integer -}

p :: Proxy 2
p = Proxy

val :: Integer
val = natVal p
```

Unable to create a value with type `t`, we are forced to pack it inside a `Proxy`. This phantom type allows us to carry around data within its type parameter, without providing it to the constructor. The actual value of the parameter is inferred by the type inferencer. `natVal` is a function provided by GHC, and allows us to turn a proxy of a natural number into an integer. The `KnownNat n` constraint tells GHC there is a way to translate `n` into an integer.

A wrapper like `Proxy` and `natVal` allow us to retrieve the length of an array solely from its type. This provides us with the tools to check the length of an array during runtime.

Let us define an `Index` class first, which serves a similar purpose to the `Ix` class used by `Data.Array`. It does however have a second purpose: to force the dimension of the type level length to match the term level indices. By adding both of them as an parameter to our `Index` class, and providing only correct instance of this class, we can force this relation:

Listing 7: Definition of `Index` and instances

```haskell
data Len a = Len

class Index n i | n -> i where
  index     :: n
  fromIndex :: n -> [i]
  size :: n -> Int
  size n = length (fromIndex n)

instance KnownNat n => Index (Len n) Int where
```

```
10   index     = Len
11   fromIndex = idxRange
12
13 instance (KnownNat m, KnownNat n)
14   => Index (Len m, Len n) (Int, Int) where
15   index = (Len, Len)
16   fromIndex (m, n) = [ (m', n') | m' <- idxRange m
17                                 , n' <- idxRange n ]
```

Line 1 We rely on our own definition of a proxy datatype, this is just so the user of Copilot can use the nicer `Len` instead of `Proxy`.

Line 3 In the definition of the class we rely on a *functional dependency*. Here we specify that `i` is dependent on `n`, that is for every `n` there is only one `i`. We need to provide this information, otherwise GHC is not able to find the correct instance of the class.

Line 4 `index` allows us to take the type literal part from our instance.

Line 5 The `fromIndex` function returns the list of indices based on the length and dimension of the array.

Lines 6 and 7 Default implementation for the length of the array.

Lines 9-11 Instance of `Index` for single dimensional arrays. As we can see, the first parameter is a natural number wrapped in a proxy, while the second argument is a single integer. This specifies that we will use integers as an index in our data. The instance forces both parameters to be of the same dimension.

We rely on the constructor of our proxy type `Len` to force the correct type on `index`, which is inferred. For the implementation of `fromIndex`, we use a simple helper function (`idxRange`) which takes a proxy containing a natural number, and returns an enumeration starting at 0.

Lines 13-17 Another instance of the class, but this time for a two dimensional array. In a similar fashion we could define instances for multiple dimensions.

The accompanying array implementation has not changed much from the simple array implementation of the previous section. The biggest difference is the incorporation of the new index type:

Listing 8: Implementation of `Array`

```
data Array n a where
  Array :: Index n i => n -> [(i, a)] -> Array n a

array :: forall n i a. Index n i => [(i, a)] -> Array n a
array xs | length xs == l = Array idx xs
         | otherwise      = error "error message" where
  idx = index
  l = size idx
```

For clarity we have added a minimalist implementation for the smart constructor as well:

Line 4 The smart constructor takes only one argument, instead of the two of the regular constructor. The argument `n` is still variable, as it depends on the length of the term level data provided to the smart constructor. Only when the full type of the array can be fully inferred, possible via other definitions, the variable is replaced by its actual value.

Lines 5 and 6 In case the length of the argument equals the length that is inferred, the array is actually created. In other cases, we can return custom error messages for specific cases.

The smart constructor ensures the length of the data equals the length specified by the type of the array. While this check is done at runtime in Haskell, it is checked before C is generated, and thus provides enough safety in the case of Copilot.

5 Code generator

Generating C code from a Copilot specification, although theoretically not very hard, is not a straightforward task. The representation of the EDSL within Copilot does not have a close relation to C. In addition, our C code comes with a number of limitations. In order for our code to pass regulations, the code must:

1. Use a constant amount of memory.
2. Be constant in execution time.
3. Not allocate memory on the heap.
4. Not use any recursive functions.
5. Be fully reentrant, that is, it can be interrupted and safely continued later on. In practice this means that it does not point to external variables.

Let us take a look at the way we represent streams in C. As it is hard to translate streams defined in Copilot directly to C, we need to come up with a rather clever way to model them.

5.1 Modelling streams in C

As we know, streams can either be read externally from the program we are monitoring, or generated from within our Copilot specification. First we will focus on generated streams, as these are the most interesting. We need to come up with a method of representing infinite streams in C. One might think that recursive functions would be a good start, but we are not allowed to use those. In addition, as we know, these might lead to stack overflows, although this depends on the implementation and compiler. In the end, the solution we took does not rely on recursive functions or loops, in fact it is actually equal to the one already used by the `Copilot-SBV` backend:

1. In C, each stream is represented by a buffer and an index. The buffer serves as a storage location for the values of the stream, and is initially filled with the values defined by the left-hand side of the `++`-operator. The index is used to store the current index in this buffer. The values inside the buffer will be overwritten over the span of the next iterations. We could say that the buffer contains the next n elements of the stream, starting at the current index.

   Dropping m elements of a stream, as described in section 3.2, is as easy as skipping the next m ones. Intuitively we now understand why we can only drop the next $m < n$ elements of a stream: the rest of the elements is not known yet, and may depend on external streams.

2. For updating the stream, we create a generator function. This function calculates the new value for the current index. Note that the generator is purely written to replace the right-hand side of `++`, as the left-hand side is already known and is used as the initial value of the buffer.

   Even if we have a definition like `s0 = [1,2] ++ s1`, the generator for `s0` will rely on `s1`'s buffer and index. This might feel counter intuitive, but later on we will take a look at an example to clarify things.

   Updating the stream is actually quite easy: for index i and buffer b, we just lookup and return $b[i]$. In case our specification uses a `drop`, we just add the drop size to the index i, and make sure we do not reach beyond the bounds of the buffer.

3. After calculating the new value, we do not write the new value to the buffer yet, because other streams may still rely on the current buffer. After every stream has been generated, we can safely write all the new values their respective current indices. For our stream this means we will be writing it to index i.

4. After saving the new value, we increase the index by one and perform a modulo operation to make sure the index does not reach beyond the last element of the buffer.

5.1.1 Code example

Here we take a look at a very basic example. For readability, we do not take a look at a complete specification, but rather at a single self recursive stream:

```
onetwothree :: Stream Int32
onetwothree = [1,2,3] ++ onetwothree
```

Judging from the definition, we can easily tell that `onetwothree` is a stream with a static part of `[1,2,3]`, and where the rest is generated by recursion, resulting in a stream of $\langle 1,2,3,1,2,3,1,2,3,\ldots\rangle$.

Listing 9: Output of `onetwothree`-example

```
 1  static int32_t s0_buff[3] = {1, 2, 3};
 2  static int32_t s0 = 1;
 3  static size_t s0_idx = 0;
 4
 5  static int32_t s0_gen () {
 6    int32_t s0_loc;
 7    {
 8      size_t idx = s0_idx;
 9      s0_loc = s0_buff[idx];
10    };
11    return s0_loc;
12  }
```

Lines 1-3 The buffer, current value and index of the stream are defined as static variables. To aid readability of code, these variables are defined globally, so there is no need to pass them around using function arguments.

Each stream, either explicitly or implicitly defined, is given a unique number by Copilot. This identifier is used internally to refer to streams, and we reuse it for the code generator. This stream has been given the name `s0`, as it currently is the only stream. Other streams may have higher numbers.

The type of the index is chosen as `size_t`. The C standard specifies that this is an integer type that is at least big enough to store a memory address. In the practically impossible event that our buffer has the size of our integrated memory, `size_t` will still be big enough to store the index of the last element of the buffer.

Line 5 The generator function for stream `s0`. It returns the new value for `s0` on the current index.

Line 6 Inside the generator function we use a local variable to store the result.

Lines 7-10 A sub scope that gives the local variable a value. Each reference to a stream in the definition of `s0`, gets its own sub scope here. The sub scope allows us to use generic variable names like `idx` or `drop` for every stream, without interfering each other, which keeps us from using prefixes. In the end having separate scopes not only make a clear distinction between streams, but also keep variable names nice and short.

Line 11 We simply return the local variable.

The generator function only calculates the new value, but does not update the buffer. In addition, the index is not updated either. Both actions are performed in a general `step`-function, which is called every iteration of our main program. In section 5.2 we will see how this function is implemented, but for now we will only take a look at the code specific to updating the buffer and index. The code should be pretty self-explanatory: it calls the generator function, stores the value in the buffer and updates the index, which loops back to `0` if it goes beyond the bounds of the buffer.

Listing 10: Code that updates buffer and index

```
s0 = s0_gen();
s0_buff[s0_idx] = s0;
++(s0_idx);
s0_idx = s0_idx % 3;
```

5.1.2 Buffer example

Although the algorithm is quite simple, it is not always easy to see why it works. For the specific example given above, where we have a single recursive stream without any operations applied, the code is easy to understand. More complicated definitions work in the same way, but are harder to understand. One of the best ways of understanding how the algorithm works, is by looking at the values of the buffers over time. Showing the contents of the buffers clearly shows where the new values are stored, and how this effects the rest of the stream.

Code listing 11 shows a small example consisting of two streams, of which one depends on the second stream, which is recursive. Figure 3 shows us to accompanying buffers over time. The figure shows us the first six steps of both streams, with the first step on the left hand side. The initial step shows us the buffers for both streams, with their initial values. A bar above a number indicates that this is the element where the current index points to.

In the first step, we see that there are two arrows: one pointing from the current element of *s0* to the current element of *s1*, and one that loops back to the current element of *s1*. These arrows show which element is read by the generator functions. In the next step, we can see that the old element has been updated with the value the arrow pointed to. In the case of *s1*, this is obviously very easy: the arrow pointed to the same element, so nothing changes. For *s0* though, it is a little bit more interesting. The arrow points to 4 from *s1*. In the second step, we see that 1 is actually replaced by 4.

Listing 11: Buffer example

```
s0 :: Stream Int32
s0 = [1,2,3] ++ s1

s1 :: Stream Int32
s1 = [4,5] ++ s1
```

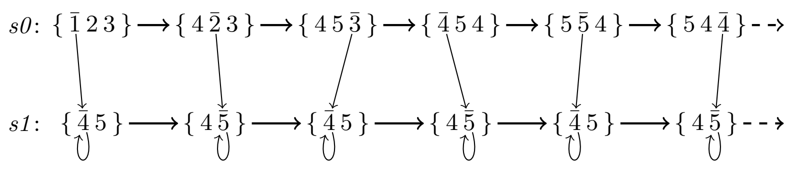

Figure 3: Contents of the buffer over time

Gradually the initial elements of *s0* are overwritten by copies of the elements of *s1*, as we expect from our stream definitions. As stated before, the bars above the elements show where the current index points to, and thus shows the current element. Listing all these elements, will yield us the values of the stream over time. From the figure we can now clearly see that *s0* $= \langle 1, 2, 3, 4, 5, 4, 5, \dots \rangle$, and that *s1* $= \langle 4, 5, 4, 5, \dots \rangle$, both according to what we expect from the definitions.

We have seen how we can represent streams in C, which is without a doubt the hardest and most important part of the code generator. There are still some other parts that need to be added to the generator: the step function, how we handle external streams, arrays and ACSL specifications. We will discuss these in the rest of this section. Note that we will not discuss structs here; structs are first-class members in C, and thus require no special attention in the code generator.

5.2 Step function

The step function is really simple and straightforward, actually we have seen most of it already in section 5.1.1. The purpose of the step function is to check if the triggers need to be fired, and update the buffers. Imagine we have a specification with two streams and a single trigger, the step function might look as follows:

Listing 12: Example of the step function

```
1  static void step () {
2      if (trigger_guard()) trigger(trigger_arg0());
3      s1 = s1_gen();
4      s0 = s0_gen();
5      s1_buff[s1_idx] = s1;
6      s0_buff[s0_idx] = s0;
7      ++(s1_idx);
8      ++(s0_idx);
9      s1_idx = s1_idx % 4;
10     s0_idx = s0_idx % 2;
11 }
```

Line 2 At first we check for every trigger if its condition is true, and execute the trigger if that is the case.

Lines 3-6 We call all generator functions, and save the outputs to the variable for that specific stream. We cannot store the result directly in a buffer, as this buffer may still be used in other generator functions. Therefore we store the value in a temporary value, and update the buffers only when all values are calculated.

Lines 7-10 After updating the buffers, it is time to update the indices as well. We increase each index by one, and then use a modulo operation to keep them between the bounds of the buffers.

5.3 External streams

External streams are rather easy to implement: we do not need to write a generator function as we can just copy the value from our main C program. We then refer to this variable when we need it. While not strictly necessary in most cases, it is a good habit to specify a type for our external values. The extern function is not able to return the correct type itself, as it knows nothing about the definition of the variable in the C sources.

External streams of arrays are identical to other types of data, with the exception that arrays are not first class in C. In C, regular data is *passed by value*, while arrays are *passed by reference* using pointer. While this works fine, it comes with the downside that this is a pointer to data in the main C program. If an interrupt occurs during monitoring and modifies this array, our monitor is not consistent anymore, which makes our monitor *non-reentrant*. This can lead to all kinds of hard to find bugs in our system. Plain datatypes and structs do not have the same problem, as these are passed value, and thus copied by default according to the C specification.

The solution is quite simple: we just need to copy the array, thus the actual data the pointer points to. We do this at the start of the step function, to ensure it happens at every iteration:

```
exarr :: Stream (Array (Len 3) Int8)
exarr = extern "exarr" Nothing
```

```
static int8_t exarr_cpy[3];

static void step () {
  memcpy(exarr_cpy, exarr, sizeof(exarr_cpy));
  ...
}
```

Now every time we want to refer to the array, we will have to refer to the copy instead. In case an interrupt will change the original array during monitoring, the monitor will catch up in the next iteration.

5.4 Arrays

The general idea of the implementation of arrays is identical to other datatypes: we use generator functions to calculate the new value of our stream and we update the buffer. As discussed in the section about external streams, arrays are not a first class citizen in C, therefore we cannot treat arrays as such. Pointers to arrays however are first class, and will be used instead of simple values. Let us take a look at a simple array example. Again we will only take a look at the global variables and generator code for now:

```
array' as = array (zip [0..] as)

arr :: Stream (Array (Len 3) Int8)
arr = [ array' [4,5,6],
        array' [1,2,3] ] ++ arr
```

We have defined a very simple array using a helper function `array'`. This function is a wrapper around the smart constructor that automatically defines the correct indices for the values given by its argument. In this case it assumes that the values are given in order, which is fine. As the definition of `arr` tells us, it is a stream consisting of two arrays, which repeats itself.

Listing 13: Example generator for arrays

```
1  static int8_t s0_buff[2][3] = {{4, 5, 6}, {1, 2, 3}};
2  static int8_t *s0 = s0_buff[0];
3  static size_t s0_idx = 0;
4
5  static int8_t *s0_gen () {
6    int8_t *s0_loc;
7    {
8      size_t idx = s0_idx;
9      s0_loc = s0_buff[idx];
10   };
11   return s0_loc;
12 }
```

Lines 1-3 The first three lines define the buffer, current value and index as we know it. Our buffer contains arrays instead of single values, and our current value is a pointer to the first element of the buffer. The length of each element, three in this case, has been taken from the type of the array.

Line 5 Our generator function now returns a pointer to the data instead of a value. As this data might change, we need to copy it to a temporary variable later on in the step function.

Lines 6-11 The body of our generator is actually identical to the definition for scalar values and structs. The only difference is that our local variable is a pointer, and we return a pointer as well.

The step function on the other hand has changed quite a bit more. Although we still run the generator, update the buffer and index, we added some intermediate copying:

Listing 14: The step function for arrays

```
1 static void step () {
2   s0 = s0_gen();
3   int8_t s0_tmp[3];
4   memcpy(s0_tmp, s0, sizeof(s0_tmp));
5   /* Other generator functions */
6   memcpy(s0_buff[s0_idx], s0_tmp, sizeof(s0_tmp));
7   ++(s0_idx);
8   s0_idx = s0_idx % 2;
9 }
```

Lines 3 and 4 The generator function returns a pointer instead of data. Normally data is copied automatically, but this pointer points to data in a buffer. The order in which generators are called, and data copied to the buffer might introduce problems. Copying the data and storing it into a temporary value before the call of a new generator, eliminates this problem.

Line 6 Here we copy the temporary data into the buffer, just as we do with scalar values.

Line 7 and 8 Just update the index as we are used to.

5.5 ACSL specifications

While the code generator produces code that is relatively simple and well tested, there is still a possibility for bugs. C is notorious for letting the user modify memory directly. This gives the programmer lots of power, but it makes it easy to introduce bugs as well. Common pitfalls in C are pointer operations, indices of arrays and problems with integer bounds.

Frama-C is a tool supporting several program analysers for C. We will be using the *WP* plugin, which verifies properties given by an ACSL specification.

Our ACSL specifications are rather simple: none of our functions take arguments and most of them do not have any side-effects. Therefore we will only use a relatively small part of the possibilities of ACSL. In this section we will discuss some of the basics we will use. For a far more in depth tutorial on Frama-C and ACSL, see *ACSL by example* [Völlinger, 2018].

ACSL code are written in C comments starting with an `@`. These comments can be placed either before a function definition, a function declaration, or as an inline comment. The specifications we will use are build around three constructs:

requires Specifies the pre-condition of a function – i.e., it must hold before the execution of the function.

ensures When we have pre-condition, we need to specify post-conditions as well. The `ensures` keyword is used to specify those.

assigns With C being a language with side-effects, we need to take these into account as well. Using multiple `assigns` definitions, we can define all side-effects.

The conditions themselves are written in a language that looks very much like C expressions: we can refer to variables and use literal values and operators. In addition we can use additional keywords that provide specific ACSL functionality. We will not list those here, as we will only use a very limited subset of them.

The C code we generate, contains basically two types of functions: generators and the step function. We will take a look at both of them.

5.5.1 Generators

Let us take a very simple Copilot specification of a counter that increases by one every iteration:

```
counter :: Stream Int8
counter = [1] ++ (counter + 1)
```

Listing 15: ACSL specification for a generator

```
1  /*@ requires \valid (s0_buff+(0..0));
2      requires 0 <= s0_idx < 1;
3      assigns  \nothing;
4      ensures  \result == ((s0_buff[s0_idx] + 1));
5  */
6  static int8_t s0_gen () {
7    int8_t s0_loc;
8    {
9      size_t idx = s0_idx;
10     s0_loc = s0_buff[idx];
11   };
12   return s0_loc + 1;
13 }
```

The resulting C code should not be a surprise, we are more interested in the ACSL specification though. Taking a closer look, we find that some of the ACSL code is not much different from the body of the function:

Line 1 The `\valid` keywords checks if its argument is a valid pointer, in our case we specify that the range `(0..0)` over `s0_buff` should be valid, that is we will not access elements beyond the specified range. Note that the range, unlike indexed arrays in C, is specified *inclusive*. In this case the index ranges from 0 to 0, so it can only be 0, which coincides with the size of the buffer.

Line 2 The second pre-condition forces the global variable `s0_idx` to be between 0 and 1. It may seem that this definition is superfluous with the presence of the first one, but it is not. Both the index and the buffer are distinct variables, that are only connected because of the way we use them. We want to be sure neither of them go beyond bounds.

Line 3 This function has no side-effects, so we assign nothing.

Line 4 In our case the post-condition is simply the result of the function. In our case, in practice the result will be semantically equal to the body of the function. This makes it easy for us to generate the ACSL code from the definition of the stream.

5.5.2 Step function

The specification for the step function is a little bit more complicated.

Listing 16: ACSL specification for the step function

```
 1  /*@ requires 0 <= s0_idx < 1;
 2        assigns  s0;
 3        assigns  s0_buff[s0_idx];
 4        assigns  s0_idx;
 5        ensures  \forall int i; 0 <= i < 1
 6            && i != \old(s0_idx) ==>
 7             s0_buff[i] == \old(s0_buff[i]);
 8  */
 9  static void step () {
10    /* Possible triggers */
11    s0 = s0_gen();
12    s0_buff[s0_idx] = s0;
13    ++(s0_idx);
14    s0_idx = s0_idx % 1;
15  }
```

Line 1 Again we want to be sure that the index variables are within bounds of the arrays they belong to.

Lines 2-4 We know that the step function updates the current value, the buffer and the index of each stream. All of these are global variables in our monitor, and therefore updating them is a side-effect of the function.

Lines 5-7 The step function is a void function and does not produce a value, therefore we do not check an equality with `\result` here. However, that does not mean we cannot express a post-condition. We know that we only update the current element in the buffer, all other elements stay untouched. With this knowledge, our post-condition is quite simple to understand: For every i in the range of the buffer and not equal to the old index, the buffer should stay the same. We make use of the `\old` keyword to denote that we want to refer to the original value of a variable and not the value after execution of the function.

6 Example

This section contains a bigger, more complete example of the code generator. The purpose is to show how all the code, including the ACSL specification, comes together. We choose not to use structs or arrays in this example, so we can make a comparison with the older back-ends. The output of the older back-ends can be found in the appendix.

The example monitor is a short but relatively complicated piece of code, using the most important constructs of Copilot. Semantically the example does not make much sense, it is just designed to give a clear example.

Listing 17: Complete Copilot example

```haskell
module Main where

import Language.Copilot
import qualified Prelude as P

{- Just one of the following back-ends -}
import Copilot.Compile.C
--import Copilot.Compile.C99
--import Copilot.Compile.SBV

temp :: Stream Int8
temp = extern "temp" Nothing

counter :: Stream Int8
counter = [1] ++ (counter + 1)

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

spec = do
  trigger "alarm" (temp > 65) [arg counter]
  trigger "fib" (fib > 1000) []

{- Compile the spec with a custom output name -}
main = do reify spec >>= compile (defaultParams {
    prefix = Just "reportexample" })
```

Listing 18: Output of example

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdint.h>

static int32_t s0_buff[2] = {1, 1};
static int8_t s1_buff[1] = {1};
static int32_t s0 = 1;
static int8_t s1 = 1;
static size_t s0_idx = 0;
static size_t s1_idx = 0;

/*@ requires \valid (s0_buff+(0..1));
    requires 0 <= s0_idx < 2;
    assigns \nothing;
    ensures \result == ((s0_buff[s0_idx] +
        s0_buff[(s0_idx+1)%2]));
*/
static int32_t s0_gen () {
  int32_t s0drop1_loc;
  int32_t s0_loc;
  {
    size_t idx = s0_idx;
    s0_loc = s0_buff[idx];
  };
  {
    size_t dropped = s0_idx + 1;
    size_t idx = dropped % 2;
    s0drop1_loc = s0_buff[idx];
  };
  return s0_loc + s0drop1_loc;
}

/*@ requires \valid (s1_buff+(0..0));
    requires 0 <= s1_idx < 1;
    assigns  \nothing;
    ensures  \result == ((s1_buff[s1_idx] + 1));
*/
static int8_t s1_gen () {
  int8_t s1_loc;
  {
    size_t idx = s1_idx;
    s1_loc = s1_buff[idx];
  };
  return s1_loc + 1;
}

/*@ requires \valid (s0_buff+(0..1));
    requires 0 <= s0_idx < 2;
    assigns  \nothing;
    ensures  \result == ((s0_buff[s0_idx] > 1000));
*/
static bool fib_guard () {
  int32_t s0_loc;
  {
    size_t idx = s0_idx;
    s0_loc = s0_buff[idx];
  };
  return s0_loc > 1000;
}

/*@ assigns  \nothing;
    ensures  \result == ((temp_cpy > 65));
*/
static bool alarm_guard () {
  return temp > 65;
}

/*@ requires \valid (s1_buff+(0..0));
    requires 0 <= s1_idx < 1;
    assigns  \nothing;
    ensures  \result == s1_buff[s1_idx];
*/
static int8_t alarm_arg0 () {
  int8_t s1_loc;
  {
    size_t idx = s1_idx;
    s1_loc = s1_buff[idx];
  };
  return s1_loc;
}

/*@ requires 0 <= s0_idx < 2;
    requires 0 <= s1_idx < 1;
    assigns  s0;
    assigns  s1;
    assigns  s0_buff[s0_idx];
    assigns  s1_buff[s1_idx];
    assigns  s0_idx;
    assigns  s1_idx;
    ensures  \forall int i; 0 <= i < 2
        && i != \old(s0_idx) ==>
         s0_buff[i] == \old(s0_buff[i]);
    ensures  \forall int i; 0 <= i < 1
        && i != \old(s1_idx) ==>
         s1_buff[i] == \old(s1_buff[i]);
*/
static void step () {
  if (fib_guard()) fib();
  if (alarm_guard()) alarm(alarm_arg0());
  s0 = s0_gen();
  s1 = s1_gen();
  s0_buff[s0_idx] = s0;
  s1_buff[s1_idx] = s1;
  ++(s0_idx);
  ++(s1_idx);
  s0_idx = s0_idx % 2;
  s1_idx = s1_idx % 1;
}
```

7 Related work

Runtime verification became an established area of research in the early 2000s. Many early efforts focused on synthesizing safety properties (informally, properties stating that "nothing bad ever happens") from temporal logic specifications. In the remainder of this section, we will briefly mention a few of the early frameworks for generating RV monitors from specifications.

Monitoring and Checking (MaC) was an early pioneering RV framework [Kim et al., 1999b, Kim et al., 1999a, Sokolsky et al., 2005]. MaC is targeted at soft real-time applications written in Java. A distinguishing feature of the MaC project is that integration and monitoring concerns are divided into separate tasks. Requirements specifications in the form of safety properties are written in the Meta Event Definition Language (MEDL). MEDL is a propositional temporal logic of events and conditions interpreted over a trace of observations of a program execution. The logic has been extended to handle dynamic indexing of properties. The Primitive Event Definition Language (PEDL) is used to define program events to be monitored and gives a mapping from the program-level events to higher-level events in the abstract specification.

Another monitoring framework, also for Java, is the Java PathExplorer (PaX) [Havelund and Roşu, 2004b, Havelund and Roşu, 2004a]. The basic architecture of PaX is similar to MaC in that it separates the integration and verification aspects of generating a monitor. PaX distinguishes itself in two areas. First, in addition to verifying logical properties, PaX performs error-pattern analysis by executing algorithms that identify error-prone programming practices. Second, the specification language is not fixed. Instead, users may define their own specification logics in Maude [Clavel et al., 1996], a language based on rewriting logics.

Monitor Oriented Programming (MOP) [Chen et al., 2004] [Chen and Roşu, 2005, Chen and Roşu, 2007] can be seen as having evolved from PaX and is based on the idea that the specification and implementation together form a system. Users provide specifications in the form of code annotations that may be written in a variety of formalisms including extended regular expressions (ERE), Java modeling language (JML), and several variants of linear temporal logic (LTL).

There are few instances of RV focused on C code. One exception is Rmor, which generates constant-memory C monitors [Havelund, 2008]. Rmor does not address real-time behavior or distributed system RV, though.

8 Discussion & Future work

The current implementation is a good start, but there are still improvements to be made. First of all, we need to find a way to actually use the accessor functions of a struct. While in Haskell we can use the functions defined by the record-syntax, we still need to translate these calls to C. Probably the easiest way to accomplish this, is by adding a new type of operator `getfield :: Struct s => (s -> a) -> s -> a` to Copilot.

This function takes an accessor function and a struct as its argument, and returns the correct field. For a simple vector struct, this may look as follows:

```
xs :: Stream Float
xs = getfield x vec
```

Here the `getfield` operator serves as a constructor for a new stream, which can then easily be translated to C.

The second shortcoming to the current implementation of structs is the overhead that writing an instance of the `Struct` and `Typed` class give. The user of Copilot needs to write an instance of both classes for his struct type. While these instances are not very hard to define, there still is a possibility of creating bugs, as there is currently no way to force the user to write a correct implementation of `toValues`. The function needs to result in a list containing the elements of the struct, with the correct type and values. Making a mistake here is easy, but due to the simplicity of the function unlikely. The problem is that there is no way for GHC to check the implementation, as lists are not a strong enough type to force the properties of `toValues`. Additionally, writing instances might give overhead, but instances can be shared among multiple projects and monitors. After working a while with several projects, a small collection of datatype definitions will be built, removing the need to write new ones after a while. We could fix both problems by relying on Template Haskell, which is a macro language for Haskell that could generate these type class instances for us, given a simpler specification of the type.

The implementation of arrays could be improved a bit as well. First of, it would be nice if we can get rid of the `Len` proxy constructor in our array type. It does not look like this is currently possible, as we need a way to construct natural numbers from term-level constructors in Haskell.

Second, it might be nicer to implement arrays in a fully dependently typed manner. This way we can force correct lengths of arrays at compile in Haskell. As we have seen, this is currently not part of Haskell, and it is unsure if this will ever change. Our current implementation has one big advantage over a dependently typed approach: it is possible to implement custom error messages in the smart constructor. If the size of the given data does not match the length of array specified by its type, we can provide the user of Copilot with a nice error message. Relying on GHC's type checker will give us the standard messages of GHC not being able to unify the two types.

Another feature still to implement is to generate a header file. Right now, only a `*.c` file is generated, which is to be included by the main program. While this is totally valid, it is good practice to only include header files in C. Generating a header file containing function and variable declarations should be easy enough, we already generate those for our `*.c` file.

Finally there is efficiency of the generated code. Currently, multiple uses of the same streams, will be calculated separately, even when the results are the same. For small specifications like our examples, this is not much of a problem, however more complicated ones might have a huge amount of duplicate executions. We need to implement an algorithm to find implicit sharing of streams, before generating any C code. When we know how streams can be shared, we are able to write more efficient code.

9 Conclusion

We have written a new C back-end for the Copilot runtime verification framework. We have taken the algorithm of `Copilot-SBV` as a starting point, but improved on it by generating more concise code. In addition, as our generator is specifically targeted to Copilot, it allows us to use some invariants on the variable and functions of the output code. We still had to identify the streams with numbers, but operations like `+` and `drop` which where saved to temporary variables, could be named accordingly. In the end, writing a custom designed code generator for Copilot, allowed us to write more concise and readable code. Choosing the variables in a way that matches its use, improved the traceability of the code as well. However, traceability could certainly be improved in the future.

Our new back-end also has added support for structs and arrays. Structs have shown to be rather easy to implement, once a suitable representation in Haskell had been found. Relying on the Haskell datatypes to model them allowed us to introduce a type-safe way of modelling C structs in Haskell. The biggest downside was that it was up to the user to write a small function to translate the datatype to a list of values, which provided the code generator with necessary information about the type.

Mistakenly, arrays seemed easier to implement at first, but proved to be a struggle to implement well. First of all, finding a suitable representation of arrays in Haskell was hard. `Data.Array` proved to lack type safety, while a type-dependent implementation was not feasible due to limitations of GHC's type inferencer. In the end a solution combining type-literals with runtime checking of length fixed the problem in a safe, but complex way. A specific `Index` class was necessary to create a connection between type level natural numbers and term level integers.

Finally we extended the back-end with ACSL specifications to aid in proving the correctness of our monitor. The specifications were added to the C code, to allow us to check the code for common mistakes like array index and integer boundary errors using the *Frama-C* tool. Generating ACSL code was surprisingly simple, but unfortunately some specifications introduced warnings in `Frama-C`, which we were unable to fix. These most likely where introduced by a bug in the tool.

The new back-end still requires some work and extra testing, before it is suitable for day-to-day use. After these fixes, it should be able to replace both `Copilot-C99` and `Copilot-SBV` successfully.

A Example code

This section shows output code of `Copilot-C99` and `Copilot-SBV`, corresponding to the example program from section 6. For brevity we only show the `*.c` files. Any other files, including `*.h` and Makefiles have been omitted.

A.1 Copilot-C99

Listing 19: Copilot-C99 output

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


#include "reportexample_copilot.h"


static uint64_t __global_clock = 0;



static const uint32_t __coverage_len = 1;
static uint32_t __coverage[1] = {0};
static uint32_t __coverage_index = 0;
struct {  /* state */
  struct {  /* reportexample_copilot */
    int32_t queue_buffer_str0[2];
    uint16_t queue_pointer_str0;
    int32_t tmp0;
    int8_t queue_buffer_str1[1];
    uint16_t queue_pointer_str1;
    int8_t tmp1;
    int8_t ext_temp;
  } reportexample_copilot;
} state =
{  /* state */
  {  /* reportexample_copilot */
    /* queue_buffer_str0 */
    { 1L
    , 1L
    },
    /* queue_pointer_str0 */  0,
    /* tmp0 */  0L,
    /* queue_buffer_str1 */
    { 1
    },
    /* queue_pointer_str1 */  0,
    /* tmp1 */  0,
    /* ext_temp */  0
  }
};

/* reportexample_copilot.sample_var_temp */
static void __r0() {
  bool __0 = true;
  int8_t __1 = temp;
  if (__0) {
    __coverage[0] = __coverage[0] | (1 << 0);
  }
  state.reportexample_copilot.ext_temp = __1;
}

/* reportexample_copilot.update_state_s0 */
static void __r3() {
  bool __0 = true;
  uint16_t __1 = state.reportexample_copilot.
      queue_pointer_str0;
  uint16_t __2 = 0;
  uint16_t __3 = __1 + __2;
  uint16_t __4 = 2;
  uint16_t __5 = __3 % __4;
  int32_t __6 = state.reportexample_copilot.
      queue_buffer_str0[__5];
  uint16_t __7 = 1;
  uint16_t __8 = __1 + __7;
  uint16_t __9 = __8 % __4;
  int32_t __10 = state.reportexample_copilot.
      queue_buffer_str0[__9];
  int32_t __11 = __6 + __10;
  if (__0) {
    __coverage[0] = __coverage[0] | (1 << 3);
  }
  state.reportexample_copilot.tmp0 = __11;
}

/* reportexample_copilot.update_state_s1 */
static void __r4() {
  bool __0 = true;
  uint16_t __1 = state.reportexample_copilot.
      queue_pointer_str1;
  uint16_t __2 = 0;
  uint16_t __3 = __1 + __2;
  uint16_t __4 = 1;
  uint16_t __5 = __3 % __4;
  int8_t __6 = state.reportexample_copilot.
      queue_buffer_str1[__5];
  int8_t __7 = 1;
  int8_t __8 = __6 + __7;
  if (__0) {
    __coverage[0] = __coverage[0] | (1 << 4);
  }
  state.reportexample_copilot.tmp1 = __8;
}

/* reportexample_copilot.fire_trigger_fib */
static void __r1() {
  int32_t __0 = 1000L;
  uint16_t __1 = state.reportexample_copilot.
      queue_pointer_str0;
  uint16_t __2 = 0;
  uint16_t __3 = __1 + __2;
  uint16_t __4 = 2;
  uint16_t __5 = __3 % __4;
  int32_t __6 = state.reportexample_copilot.
      queue_buffer_str0[__5];
  bool __7 = __0 < __6;
  if (__7) {
    reportexample_fib();
    __coverage[0] = __coverage[0] | (1 << 1);
  }
}

/* reportexample_copilot.fire_trigger_alarm */
static void __r2() {
  int8_t __0 = 65;
  int8_t __1 = state.reportexample_copilot.ext_temp;
  bool __2 = __0 < __1;
  uint16_t __3 = state.reportexample_copilot.
      queue_pointer_str1;
  uint16_t __4 = 0;
  uint16_t __5 = __3 + __4;
  uint16_t __6 = 1;
  uint16_t __7 = __5 % __6;
  int8_t __8 = state.reportexample_copilot.
      queue_buffer_str1[__7];
  if (__2) {
    reportexample_alarm(__8);
    __coverage[0] = __coverage[0] | (1 << 2);
  }
}

/* reportexample_copilot.update_buffer_s0 */
static void __r5() {
  bool __0 = true;
  int32_t __1 = state.reportexample_copilot.tmp0;
  uint16_t __2 = state.reportexample_copilot.
      queue_pointer_str0;
  uint16_t __3 = 1;
  uint16_t __4 = __2 + __3;
  uint16_t __5 = 2;
  uint16_t __6 = __4 % __5;
  if (__0) {
    __coverage[0] = __coverage[0] | (1 << 5);
  }
  state.reportexample_copilot.queue_buffer_str0[__2] =
      __1;
  state.reportexample_copilot.queue_pointer_str0 = __6;
}

/* reportexample_copilot.update_buffer_s1 */
static void __r6() {
  bool __0 = true;
  int8_t __1 = state.reportexample_copilot.tmp1;
  uint16_t __2 = state.reportexample_copilot.
      queue_pointer_str1;
  uint16_t __3 = 1;
  uint16_t __4 = __2 + __3;
  uint16_t __5 = __4 % __3;
  if (__0) {
    __coverage[0] = __coverage[0] | (1 << 6);
  }
  state.reportexample_copilot.queue_buffer_str1[__2] =
      __1;
  state.reportexample_copilot.queue_pointer_str1 = __5;
}


static void __assertion_checks() {
}


void reportexample_copilot()
{

  {
    static uint8_t __scheduling_clock = 0;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r0();  /*
          reportexample_copilot.sample_var_temp */
      __scheduling_clock = 7;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
  {
    static uint8_t __scheduling_clock = 4;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r3();  /*
          reportexample_copilot.update_state_s0 */
      __assertion_checks(); __r4();  /*
          reportexample_copilot.update_state_s1 */
      __scheduling_clock = 7;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
  {
    static uint8_t __scheduling_clock = 5;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r1();  /*
          reportexample_copilot.fire_trigger_fib */
      __assertion_checks(); __r2();  /*
          reportexample_copilot.fire_trigger_alarm */
      __scheduling_clock = 7;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
  {
    static uint8_t __scheduling_clock = 7;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r5();  /*
          reportexample_copilot.update_buffer_s0 */
      __assertion_checks(); __r6();  /*
          reportexample_copilot.update_buffer_s1 */
      __scheduling_clock = 7;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }

  __global_clock = __global_clock + 1;

}

void reportexample_step()
{
  reportexample_copilot();reportexample_copilot();
      reportexample_copilot();reportexample_copilot();
      reportexample_copilot();reportexample_copilot();
      reportexample_copilot();reportexample_copilot();
}
```

A.2 Copilot-SBV

Listing 20: reportexample_driver.c

```c
/* Driver for SBV program generated from Copilot. */
/* Edit as you see fit */

#include "reportexample_internal.h"
#include "reportexample_copilot.h"

/* Observers */


/* Variables */
static SInt32 tmp_0 = 1;
static SInt8 tmp_1 = 1;
static SInt32 queue_0[2] = { 1, 1 };
static SInt8 queue_1[1] = { 1 };
static SWord32 ptr_0 = 0;
static SWord32 ptr_1 = 0;
static SInt8 ext_temp = 0;

void static sampleExts(void) {
  ext_temp = temp;
}

void static fireTriggers(void) {
  if (trigger_guard_alarm(ext_temp))
    alarm(trigger_alarm_arg_0(queue_1, ptr_1));
  if (trigger_guard_fib(queue_0, ptr_0))
    fib();
}

/*@
 assigns \nothing;
 */
void static updateObservers(void) {
}

/*@
 assigns tmp_0;
 assigns tmp_1;
 */
void static updateStates(void) {
  tmp_0 = update_state_0(queue_0, ptr_0);
  tmp_1 = update_state_1(queue_1, ptr_1);
}

/*@
 assigns queue_0[ptr_0];
 ensures queue_0[ptr_0] == tmp_0;
 assigns queue_1[ptr_1];
 ensures queue_1[ptr_1] == tmp_1;
 */
void static updateBuffers(void) {
  queue_0[ptr_0] = tmp_0;
  queue_1[ptr_1] = tmp_1;
}

/*@
 assigns ptr_0;
 ensures ptr_0 == (\old (ptr_0 ) + 1) % 2;
 assigns ptr_1;
 ensures ptr_1 == (\old (ptr_1 ) + 1) % 1;
 */
void static updatePtrs(void) {
  ptr_0 = (ptr_0 + 1) % 2;
  ptr_1 = (ptr_1 + 1) % 1;
}
/* Idents */

/*@
 assigns \nothing;
 */
SBool ident_bool(SBool a) {return a;}
/*@
 assigns \nothing;
 */
SWord8 ident_word8(SWord8 a) {return a;}
/*@
 assigns \nothing;
 */
SWord16 ident_word16(SWord16 a) {return a;}
/*@
 assigns \nothing;
 */
SWord32 ident_word32(SWord32 a) {return a;}
/*@
 assigns \nothing;
 */
SWord64 ident_word64(SWord64 a) {return a;}
/*@
 assigns \nothing;
 */
SInt8 ident_int8(SInt8 a) {return a;}
/*@
 assigns \nothing;
 */
SInt16 ident_int16(SInt16 a) {return a;}
/*@
 assigns \nothing;
 */
SInt32 ident_int32(SInt32 a) {return a;}
/*@
 assigns \nothing;
 */
SInt64 ident_int64(SInt64 a) {return a;}
/*@
 assigns \nothing;
 */
SFloat ident_float(SFloat a) {return a;}
/*@
 assigns \nothing;
 */
```

```c
SDouble ident_double(SDouble a) {return a;}

void reportexample_step(void) {
  sampleExts();
  fireTriggers();
  updateObservers();
  updateStates();
  updateBuffers();
  updatePtrs();
}

void reportexample_testing(void) {
  for(;;) reportexample_step();
}
```

Listing 21: trigger_alarm_arg_0.c

```c
/* File: "trigger_alarm_arg_0.c". Automatically generated
     by SBV. Do not edit! */

#include "reportexample_internal.h"

/* User given declarations: */
/*test 003*/
/*@
 assigns \nothing;
 ensures \abs(\result - queue_1[0]) <= 0.1;
*/
SInt8 trigger_alarm_arg_0(const SInt8 *queue_1,
                          const SWord32 ptr_1)
{
  const SInt8   s0 = queue_1[0];
  return s0;
}
```

Listing 22: trigger_guard_alarm.c

```c
/* File: "trigger_guard_alarm.c". Automatically generated
     by SBV. Do not edit! */

#include "reportexample_internal.h"

/* User given declarations: */
/*test 006*/
/*@
 assigns \nothing;
 ensures \result == ((ext_temp > 65));
*/
SBool trigger_guard_alarm(const SInt8 ext_temp)
{
  const SInt8 s0 = ext_temp;
  const SBool s2 = s0 > 65;

  return s2;
}
```

Listing 23: trigger_guard_fib.c

```
/* File: "trigger_guard_fib.c". Automatically generated
    by SBV. Do not edit! */

#include "reportexample_internal.h"

/* User given declarations: */
/*test 006*/
/*@
 assigns \nothing;
 ensures \result == ((queue_0[ptr_0] > 1000));
*/
SBool trigger_guard_fib(const SInt32 *queue_0, const
    SWord32 ptr_0)
{
  const SInt32  s0 = queue_0[0];
  const SInt32  s1 = queue_0[1];
  const SWord32 s2 = ptr_0;
  const SInt32 table0[] = {
      s0, s1
  };
  const SWord32 s4 = (0x00000002UL == 0) ? s2 : (s2 % 0
      x00000002UL);
  const SInt32  s5 = table0[s4];
  const SBool   s7 = s5 > 0x000003e8L;

  return s7;
}
```

Listing 24: update_state_0.c

```
/* File: "update_state_0.c". Automatically generated by
    SBV. Do not edit! */

#include "reportexample_internal.h"

/* User given declarations: */
/*test 001*/
/*DotBegin
digraph G {
node [shape=box]

0 [label="file:
?????",color=red, style=filled]
1 [label="op2: +",color=green4, style=filled]
0 -> 1
2 [label="stream: 0",color=crimson, style=filled]
1 -> 2
3 [label="drop 1:
stream: 0",color=crimson, style=filled]
1 -> 3


}

DotEnd*/
/*@
```

```
 assigns \nothing;
 ensures \abs(\result - ((queue_0[ptr_0] + queue_0[(ptr_0
      + 1) % 2]))) <= 0.1;
*/
SInt32 update_state_0(const SInt32 *queue_0, const
    SWord32 ptr_0)
{
  const SInt32  s0 = queue_0[0];
  const SInt32  s1 = queue_0[1];
  const SWord32 s2 = ptr_0;
  const SInt32 table0[] = {
      s0, s1
  };
  const SWord32 s4 = (0x00000002UL == 0) ? s2 : (s2 % 0
      x00000002UL);
  const SInt32  s5 = table0[s4];
  const SWord32 s7 = s2 + 0x00000001UL;
  const SWord32 s8 = (0x00000002UL == 0) ? s7 : (s7 % 0
      x00000002UL);
  const SInt32  s9 = table0[s8];
  const SInt32  s10 = s5 + s9;

  return s10;
}
```

Listing 25: update_state_1.c

```
/* File: "update_state_1.c". Automatically generated by
    SBV. Do not edit! */

#include "reportexample_internal.h"

/* User given declarations: */
/*test 001*/
/*DotBegin
digraph G {
node [shape=box]

0 [label="file:
?????",color=red, style=filled]
1 [label="op2: +",color=green4, style=filled]
0 -> 1
2 [label="stream: 1",color=crimson, style=filled]
1 -> 2
3 [label="const: 1",color=red1, style=filled]
1 -> 3


}

DotEnd*/
/*@
 assigns \nothing;
 ensures \abs(\result - ((queue_1[0] + 1))) <= 0.1;
*/
SInt8 update_state_1(const SInt8 *queue_1, const SWord32
    ptr_1)
```

```
{
  const SInt8    s0 = queue_1[0];
  const SInt8    s3 = s0 + 1;

  return s3;
}
```

References

[SAE, 1996] (1996). SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.

[Baudin et al., 2015] Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V. (2015). *ACSL: ANSI/ISO C Specification Language, version 1.10*.

[Bertot and Castran, 2010] Bertot, Y. and Castran, P. (2010). *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated.

[Butler, 2008] Butler, R. W. (2008). A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center.

[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.

[Chen et al., 2004] Chen, F., D'Amorim, M., and Roşu, G. (2004). A formal monitoring-base framewrok for software development analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, LNCS, pages 357–373. Springer.

[Chen and Roşu, 2005] Chen, F. and Roşu, G. (2005). Java-MOP: a monitoring oriented programming environment for Java. In *11th Intl. Conf. on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer.

[Chen and Roşu, 2007] Chen, F. and Roşu, G. (2007). MOP: an efficient and generic runtime verification framework. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 569–588.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM.

[Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[Clavel et al., 1996] Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. (1996). Principles of maude.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY.

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

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32.

[Frama-C webpage, 2018] Frama-C webpage (Accessed May 2018). Frama-C. Accessed November, 2018. `http://frama-c.com/index.html`.

[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.

[Halbwachs et al., 1993] Halbwachs, N., Lagnier, F., and Raymond, P. (1993). Synchronous observers and the verification of reactive systems. In *Third International Conference on Algebraic Methodology and Software Technology*, pages 83–96. Springer Verlag.

[Havelund, 2008] Havelund, K. (2008). Runtime verification of C programs. In *Testing of Software and Communicating Systems (TestCom/-FATES)*, pages 7–22. Springer.

[Havelund and Roşu, 2004a] Havelund, K. and Roşu, G. (2004a). Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2).

[Havelund and Roşu, 2004b] Havelund, K. and Roşu, G. (2004b). An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215.

[Hawkins, 2008] Hawkins, T. (2008). Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP). Available at `http://cufp.galois.com/2008/schedule.html`.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Kim et al., 1999a] Kim, M., Sokolsky, O., and Viswanathan, M. (1999a). Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 279–287.

[Kim et al., 1999b] Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. (1999b). Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems*, pages 114–122.

[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.

[Laprie, 1995] Laprie, J.-C. (1995). Dependability—its attributes, impairments and means. In *Predictability Dependable Computing Systems*, pages 3–24. Springer.

[Leveson, 2012] Leveson, N. G. (2012). *Engineering a Safer World: Systems Thinking Applied to Safety.* MIT Press.

[Owre et al., 1992] Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *cade92*, volume 607 of *lnai*, pages 748–752. SV.

[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., 2011] Pike, L., Niller, S., and Wegmann, N. (2011). Runtime verification for ultra-critical systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, LNCS. Springer.

[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).

[RTCA, 2011] RTCA (2011). Formal methods supplement to do-178c and do-278a. RTCA, Inc. RCTA/DO333.

[RTCA, 2011] RTCA (2011). Software considerations in airborne systems and equipment certification. RTCA, Inc. RCTA/DO-178C.

[Sha, 2001] Sha, L. (2001). Using simplicity to control complexity. *IEEE Software*, pages 20–28.

[Sokolsky et al., 2005] Sokolsky, O., Sammapun, U., Lee, I., and Kim, J. (2005). Run-time checking of dynamic properties. 144(4):91–108.

[Völlinger, 2018] Völlinger, A. C. R. C. L. G. K. H. T. L. H. W. P. J. S. K. (2018). Acsl by example.

[Wegmann et al., 2015] Wegmann, N., Pike, L., and Niller, S. (2015). An introduction to copilot.