Runtime Verification with the Copilot Language

A Hands on Introduction

Frank Dedden

System F Computing

March 15, 2024

Runtime Verification

Introduction

Ultra-critical systems, e.g.:

- Civil aircraft
- Power plants
- Storm surge barriers

Assurance levels: $< 10^{-9}/\mathrm{h}$ for civil aircraft.

High complexity, consisting of:

- Mechanics
- Electronics / avionics
- Software
- ...

How to achieve these requirements?

- Mechanics: Use models, simulations, structural testing.
- Electronics: Formal verification, simulations, testing.
- Software: Formal verification, (automated) testing.

How to achieve these requirements?

- Mechanics: Use models, simulations, structural testing.
- Electronics: Formal verification, simulations, testing.
- Software: Formal verification, (automated) testing.

In practice, this is not enough:

- Testing does not cover all the cases.
- Too complex for complete formal verification.
- Hardware can fail in unpredictable ways.
- Generally impossible to address all possible failures.

Solution

Runtime Verification!

- A ‘lightweight’ verification approach.
- Check properties while system is in use.
- On violation of a property, system can react.

Challenges

- Hard-realtime execution.
- Predictable memory usage.
- Minimal interference with main system.

Problem

A low-level language seems obvious.
What if runtime verification itself becomes too complex?

Problem

A low-level language seems obvious.
What if runtime verification itself becomes too complex?

Solution

Use a domain specific language.

Copilot

Overview

- A stream-based hard-realtime programming language.
- Originally developed by NASA and the NIA since 2010.
- Currently maintained by NASA, with help from Galois Inc. and external contributors.
- Free and open-source under BSD-3 license.

NASA classification

- **Class D, Basic**: Science/Engineering Design and Research and Technology Software.
- **Class C, Mission Support**: Software or Aeronautic Vehicles, or Major Engineering/Research Facility Software.

Components

A modular design of multiple components (among others):

- **copilot-core**: Intermediate representation of Copilot programs.
- **copilot-language**: Contains the language front-end.
- **copilot-libraries**: A set of utility functions for Copilot.
- **copilot-c99**: A back-end for Copilot targeting C99.
- **copilot-bluespec**: Generates Bluespec FPGA code.
- **copilot-theorem**: Extensions for proving properties.

Streams

$s \rightsquigarrow \{1, 2, 3, 4, 5, \ldots\}$

- Values that change over time.
- Conceptually similar to infinite lazy lists.
- In Copilot all streams of a program advance at the same moment.

Language basics

```
s0 :: Stream Bool
s0 = constant True        -- s0 ~> {True, True, True, ...}

s1 :: Stream Int32
s1 = 6                    -- s1 ~> {6, 6, 6, ...}
```

The boolean streams `true` and `false` are predefined:

```
true  :: Stream Bool
false :: Stream Bool
```

Operators

Mathematical operators are point-wise applied:

```
x :: Stream Int32
x = 5 + 5                 -- x ~> {10, 10, 10, ...}

y :: Stream Int32
y = x * x                 -- y ~> {100, 100, 100, ...}

z :: Stream Bool
z = x == 10 && y < 200    -- z ~> {True, True, True, ...}
```

Appending values

Streams can change over time, by appending values:

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

w :: Stream Int32
w = [5, 6, 7] ++ x        -- w ~> {5, 6, 7, 10, 10, ...}
```

Note

- The numbers between `[]` are not streams, but values!
- For the Haskellers: The type of `(++)` actually contains the `Typed a` constraint, this is left out for brevity here.

The definitions can be self-recursive:

```
counter :: Stream Int64
counter = [0] ++ counter + 1  -- counter ~> {0, 1, 2, 3, ...}
```

Dropping values

The counterpart of ++ is `drop`:

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

numbers :: Stream Int64
numbers = [1,34,2,9,8,15] ++ numbers

numbers' :: Stream Int64
numbers' = drop 2 numbers    -- numbers' ~> {2, 9, 8, 15, ...}
```

Note

Dropping values is only possible on streams where enough values are appended prior.

External values

Copilot can interface with existing C code by reading a variable as a stream.

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

t :: Stream Double
t = extern "temperature" Nothing
```

Practical session

Practical session

Repository with a Docker file and exercises:

```
$ git clone https://github.com/fdedden/bobkonf-2024-copilot-tutorial
$ cd bobkonf-2024-copilot-tutorial
bobkonf-2024-copilot-tutorial$ make build
```

For this session, Copilot will be used inside the Docker container.

Thank you for your attention

- Copilot website: https://copilot-language.github.io
  - The Copilot manual.
  - Links to source code.
- Copilot 3 Technical Report: https://ntrs.nasa.gov/citations/20200003164
- Ivan Perez and Frank Dedden, 2023, *The Essence of Reactivity*: https://doi.org/10.1145/3609026.3609727
- Questions / Contributions: https://github.com/Copilot-Language
- Contact: frank@systemf.dev
- ORCID: 0009-0000-9311-5787

Airplane picture in animation designed by macrovector / Freepik