Monitoring Distributed Hard Real-Time Systems

Lee Pike
Galois, Inc.
leepike@galois.com

Alwyn Goodloe
National Institute of Aerospace
Alwyn.Goodloe@nianet.org

This research is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office. Dr. Ben Di Vito of NASA Langley advises this research.

PROBLEM

Can we make safety-critical embedded systems ultra-reliable?

Modern aircraft and manned spacecraft rely on **complex** embedded software for guidance, navigation, and control (GN&C). Software failures can lead to the loss of the vehicle and human life.

Characteristics of safety-critical GN&C software:

- Distributed and redundant (for fault-tolerance)
- Hard real-time (i.e., constant time usage)
- Constant memory usage (i.e., no dynamic memory allocation)
- (Usually) C source code

STRATEGY

Runtime monitoring

Safety-critical software is rigorously designed using regimented coding standards, testing standards, and **formal methods** to increase confidence in its correctness. But these do not *guarantee* correctness at runtime.

NASA is supporting research into **runtime monitoring**, where *a system is observed in operation by a monitor to check conformance to safety-properties*. A monitor must be simpler than the software monitored.

Research Challenge

Runtime monitoring for general-purpose software in high-level languages (e.g., Java) is a mature field. We are researching how to adapt runtime monitoring to safety-critical GN&C software with the named characteristics.

A fundamental challenge our research addresses is **compositionality**: how do monitors and the observed software interact?

- ***Functionality*:** monitors must not change the functionality of the observed system, unless it violates its specification.
- ***Schedulability/Time*:** monitors must execute in constant time and not interfere with the timing properties of the observed program.
- ***Reliability*:** the reliability of the monitor + observed system is greater or equal to the reliability of the observed system alone.
- ***Certification*:** Monitors should require very few---if any---modifications to the source code of the observed programs.

APPROACH

Synthesizing monitors

Our approach is to **automatically synthesize** monitors from high-level specifications. The **inputs** are

- A set of **state variables** to observe.
- A set of **monitor variables** acting as "history variables" for observed state variables.
- A **schedule** for (1) when to observe state variables and (2) when to check in variants.
- A **distribution configuration** describing how monitors should be distributed or centralized.

The **output** is an embedded C source files meeting the constraints. In a distributed mode, a set of C source files are generated. The monitors themselves are also hard real-time and execute in constant memory.

Atom/Haskell specification

```
gcd:: Atom ()
gcd = do
  -- External reference to value A.
  a <- word32' "a"
  -- External reference to value B.
  b <- word32' "b"

  -- The external running flag.
  running <- bool' "running"

  -- A rule to modify A.
  atom "a_minus_b" $ do
    cond $ value a >. value b
    a <== value a - value b

  -- A rule to modify B.
  atom "b_minus_a" $ do
    cond $ value b >. value a
    b <== value b - value a

  -- A rule to clear the running flag.
  atom "stop" $ do
    cond $ value a ==. value b
    running <== false
```

Synthesized embedded C code

```
      . . .

/* example.a_minus_b */
static void __r0(void) {
  unsigned long int __0 = b;
  unsigned long int __1 = a;
  unsigned char __2 = __0 < __1;
  unsigned long int __3 = __1 - __0;
  unsigned long int __4 = __2 ? __3 : __1;
  a = __4;
}

void example(void) {
  if (__clock % 1 == 0) {
    __r0();  /* example.a_minus_b */
    __r1();  /* example.b_minus_a */
    __r2();  /* example.stop */
  }

  __clock = __clock + 1;
}
      . . .
```

Credit: Tom Hawkins.

TOOLS

Code Synthesis

A correct-by-construction approach generates low-level C code from high-level models. Commercial code-generators like Simulink/Stateflow and SCADE can generate embedded C from high-level models, but they are closed-source so are not extensible.

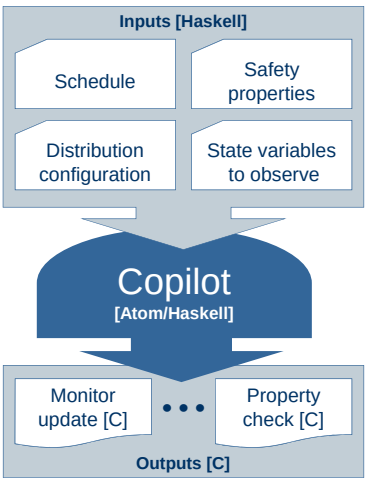
Customized Code Generation

Atom is an extensible, open-source, domain-specific library for the Haskell functional programming language. The library rewrites an Atom/Haskell program to embedded C source files. Eaton, Ltd. has used Atom to synthesize control systems for commercial vehicles. Atom-synthesized C is *guaranteed* to have deterministic memory usage and timing. Atom is *not* a new compiler so can reuse all of the language infrastructure provided by Haskell---thus, we get *cheap, customized, embedded code generation*!

PROTOTYPE

Copilot

Copilot synthesizes monitors for safety-critical GN&C software. Copilot is built on top of Atom and is an extensible, open-source set of libraries for specifying and synthesizing monitors, *without* requiring the development of a new languages or compiler infrastructure.



LEARN MORE

Further Reading

1. Alwyn Goodloe and Lee Pike. *Monitoring distributed real-time systems: a survey and future directions*. To be published as a NASA Contractor Report.
2. Lee Pike, Geoffrey M. Brown, and Alwyn Goodloe. *Roll your own test bed for embedded real-time protocols: a Haskell experience*. In the Haskell Symposium, 2009.
3. Alwyn Goodloe and Lee Pike. *Toward monitoring fault-tolerant embedded systems (extended abstract)*. In the International Workshop on Software Health Management (SHM'09), 2009.