Real-Time Control With the Sparse Synchronous Model

Stephen A. Edwards

The Problem

Rats: A Motivating Example

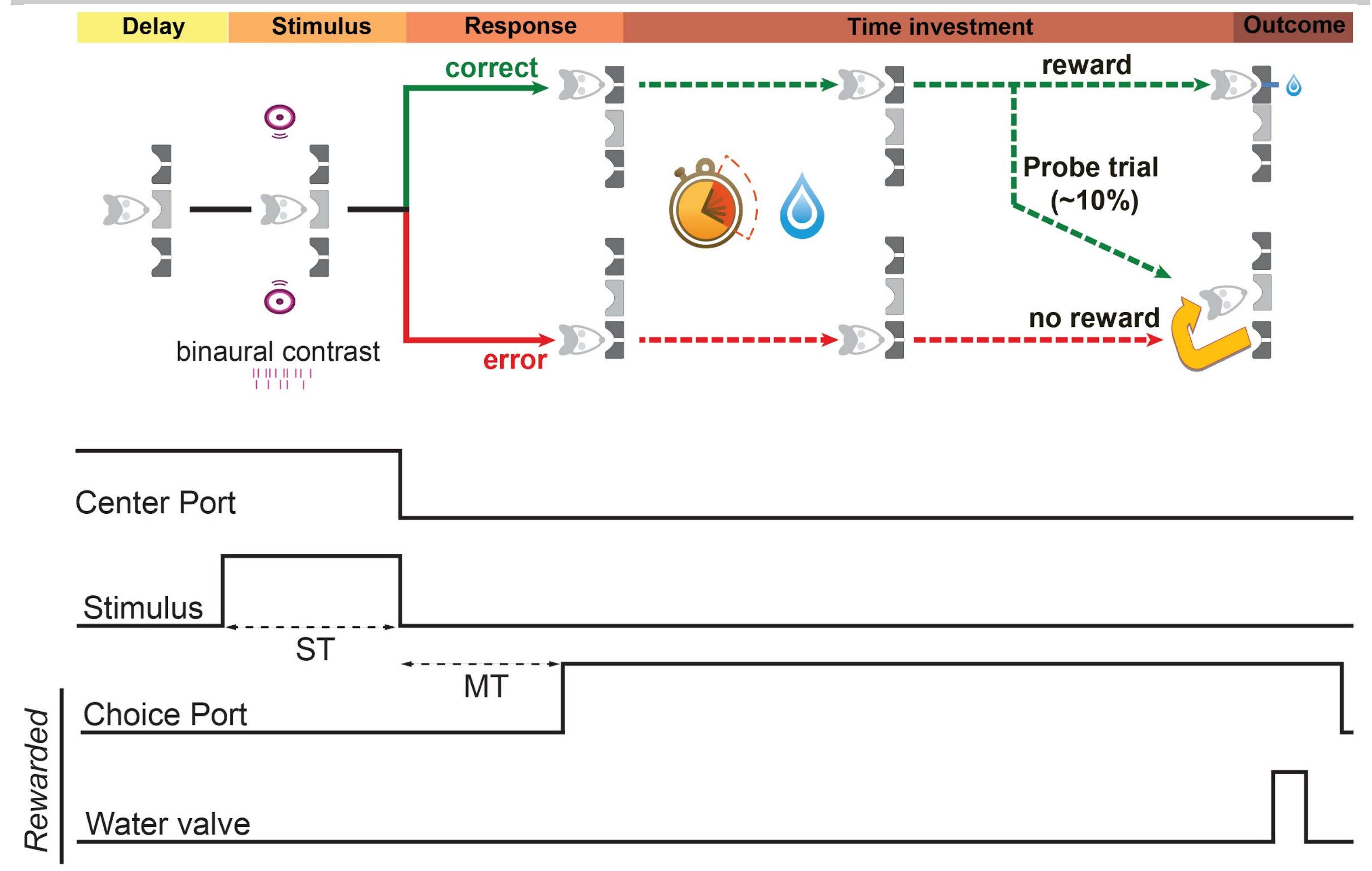
A colleague trains rats on various tasks and analyzes their response times.

Such experiments typically use a microcontroller to generate stimulus and collect data; solutions usually ad hoc.

It's difficult for other researchers to reproduce the results.

He wanted a language to precisely specify these experiments: what stimulus is presented to the subjects and their responses, carefully timed.

A Sample Task (or, How Biologists "Program")



Challenges

C, the typical language available on microcontrollers, has no notion of time

Library functions like `sleep()` tend to be imprecise

Real-Time Operating Systems (RTOSes) are a step in the right direction, but do not provide precise timing control

Microcontrollers have timers accurate to at least 1 μs, but they are complicated to use

Our goal: an easy-to-use programming language that provides precise, reproducible timing control

Our Solution

We developed a runtime system able to provide precise timing control.

Here are examples it can run, expressed in the toy language we used for experiments.

A traditional imperative language...

```
gcd(a, b, &r)
  while a != b
    if  a < b then
      b = b − a
    else
      a = a − b
  r = a
```

Named routines, no return values

Pass-by-value (integer) arguments

Pass-by-reference arguments

Imperative *while* loops

Conditionals

Imperative assignment

Assignment to a reference returns a value

...with concurrent function calls...

```
foo(&a)
  a = a + 2

bar(&a)
  a = a ∗ 4

main()
  var a = 1
  fork foo(a) bar(a)
  // a = 12 = (1 + 2) * 4 here
  fork bar(a) foo(a)
  // a = 50 = (12 * 4) + 2 here
```

Concurrent recursive calls

Concurrently running routines may interfere

Deterministic: execution order prescribed by call order

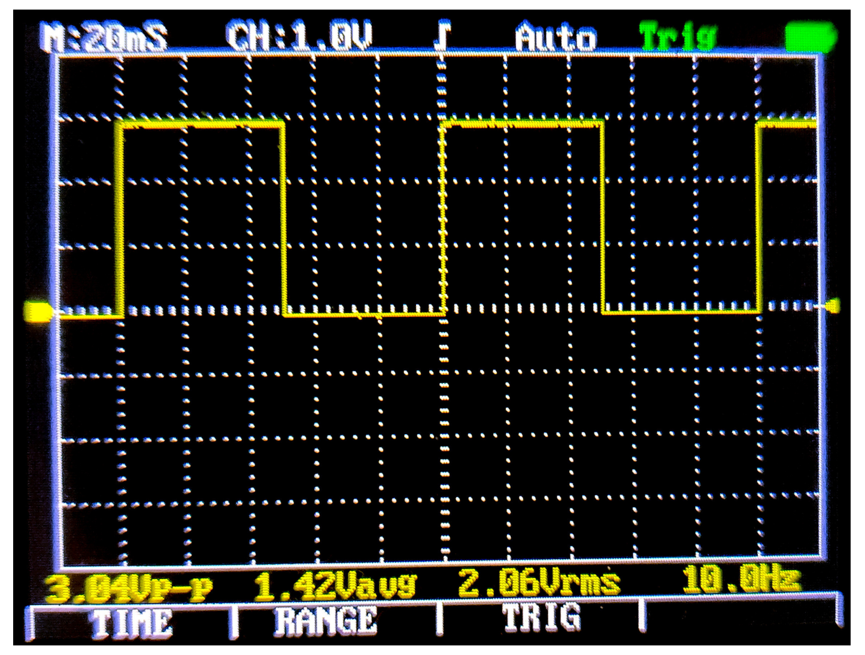
No true parallelism, for now

...and precise timing specification

```
blink(&led)
  while 1
    after 50 ms led = 1
    wait led
    after 50 ms led = 0
    wait led
```

Delayed assignment: future update scheduled

Blocking wait-for-write



Details

Main Features

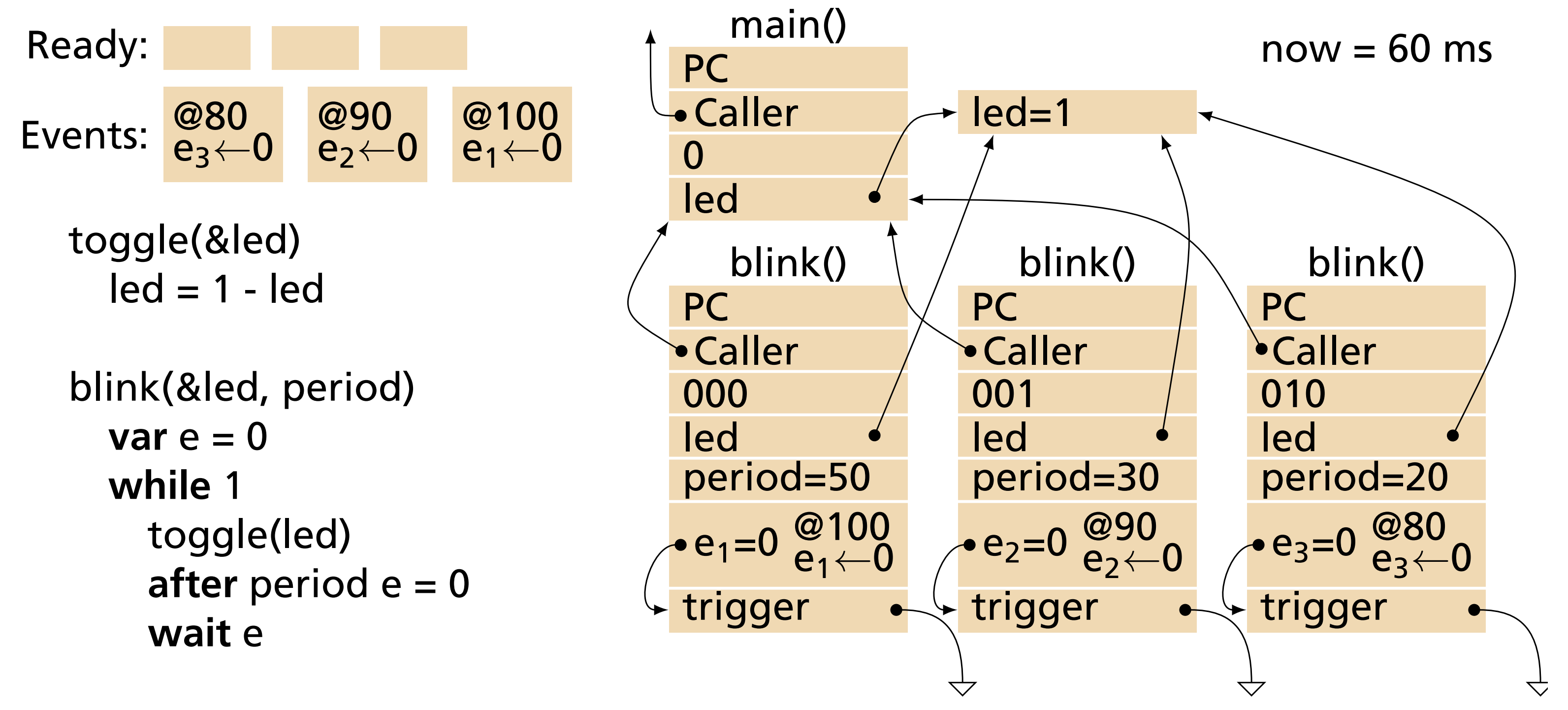
Concurrently running tasks (`fork`) that can block on variable updates (`wait`)

Can schedule a future variable update (`after`)

The program sees and controls "model time," like it was running on an infinitely fast processor.

Requested delays independent of processor speed, controlled by microcontroller timer, not processor execution speed.

Runtime System



```
toggle(&led)
  led = 1 - led

blink(&led, period)
  var e = 0
  while 1
    toggle(led)
    after period e = 0
    wait e

main(&led)
  fork blink(led, 50ms) blink(led, 30ms) blink(led, 20ms)
```

Built around two queues:

The Ready Queue holds routines that are yet to be executed at the current time, prioritized according to their position in the program to ensure determinism

The Event Queue holds future variable updates, prioritized by time

In an instant, the runtime updates variables with events, then runs all the routines this has triggered, which may trigger more routines and schedule future events.

After the runtime is done for an instant, it sets a timer to wait itself up for the next event.

Environmental inputs may wake it up sooner.

Ongoing Work

We are developing a language and compiler targeting the Sparse Synchronous Model and its runtime system

We want to port the runtime to a variety of microcontrollers to give researchers a choice