Creating a Language for Writing Real-Time Applications for the Internet of Things

Robert Krook*, John Hui†, Bo Joel Svensson*, Stephen A. Edwards†, and Koen Claessen*
*Chalmers University of Technology, Gothenburg, Sweden
†Columbia University, New York, USA
Email: krookr@chalmers.se, j-hui@cs.columbia.edu, joels@chalmers.se, sedwards@cs.columbia.edu, and koen@chalmers.se

***Abstract*—We describe the development of a new programming language Scoria and its compiler. Scoria is a high-level reactive real-time language based on the sparse synchronous model (SSM), designed to produce time- and power-efficient low-level C code that can run on small IoT devices. While the compiler is not yet in a state where it is meaningful to measure power usage, we carefully profile the timing behaviour and identify bottlenecks that can improve performance. The language and compiler are implemented as an Embedded Domain-Specific Language (EDSL) on top of Haskell.**

***Index Terms*—Real-time, IoT, Compilers, Embedded Domain-Specific Languages**

I. Introduction

Devices for the Internet of Things (IoT) typically contain hardware for sensors, actuators, and wireless communication, and often need to run on batteries whose life expectancy is a couple of years. These peripherals and resources need to be carefully managed in order to maximize lifetime and reliability. The devices are typically programmed in C because of their low-level nature, timing, and power requirements. They are coded in a *reactive* style as the software needs to react to external events, such as a triggered sensor. Such programs are often written to register a myriad of callback functions to handle these events and communicate with other parts of the program. The resulting asynchronous code is often error-prone and hard to maintain, a situation often dubbed "callback hell."

Our language, Scoria, eschews callbacks in favor of lightweight threads that can block on external and internal events and provides precise control over the time at which code executes. We embed [1] Scoria in Haskell—which explains its syntactic idiosyncrasies—to enable rapid development of its compiler, which generates C code that uses our runtime system (RTS) to interface with hardware timers and other peripherals.

Figure 1 is a simple Scoria program: a remote-controlled square wave signal generator whose frequency can be adjusted by Bluetooth Low Energy (BLE) radio packets. The `entry` routine (lines 15–18) expects a BLE receiver and GPIO pin in the environment (line 15), creates a shared variable `hperiod`, and runs `sigGen` and `remoteControl` concurrently (line 18).

The `sigGen` routine (lines 1–4) generates a precisely timed square wave on the GPIO pin by scheduling future output events and blocking on them. It takes a reference to the `hperiod` variable and enters an infinite loop that schedules a future toggle update to the GPIO pin (line 3), then blocks until the update occurs (line 4). The delay between toggle events is exactly the *model time* set by the `hperiod` variable, regardless of the loop's execution time—Scoria's model time only advances at blocking statements like `wait`. The `after` directive schedules the update according to model time.

```
1  sigGen :: (?out0 :: Ref GPIO) => Ref Word64 -> SSM ()
2  sigGen hperiod = routine $ while true (do
3    after (ns (deref hperiod)) ?out0 (not' (deref ?out0))
4    wait ?out0)
5
6  remoteControl :: (?ble :: BLE) => Ref Word64 -> SSM ()
7  remoteControl hperiod = routine $ do
8    enableScan ?ble
9    while true (do
10     wait (scanref ?ble)
11     if deref (scanref ?ble) ==. 0
12       then hperiod <~ deref hperiod * 2
13       else hperiod <~ max' (deref hperiod /. 2) 1)
14
15 entry :: (?ble :: BLE, ?out0 :: Ref GPIO) => SSM ()
16 entry = routine $ do
17   hperiod <- var (time2ns (secs 1))
18   fork [sigGen hperiod, remoteControl hperiod]
```

Fig. 1: A remote-control signal generator in Scoria. `sigGen` produces a square wave on the `out0` GPIO pin that cycles every two `hperiod` nanoseconds. When a Bluetooth packet arrives, `remoteControl` doubles or halves the period.

The `remoteControl` routine (lines 6–13) waits for a BLE packet (line 10), and doubles or halves `hperiod` in response (lines 12 and 13). The `enableScan` call (line 8) enables the Bluetooth receiver, which then generates events on the variable given by `scanref ?ble`.

Figure 2 shows code that runs at compile time to initialize the signal generator program's environment. References to a GPIO pin and the BLE controller are obtained in lines 3 and 4, and added to the environment (indicated by the question marks) in lines 5 and 6. In lines 7–9, the SSM ready queue is populated with the `entry` routine and I/O handlers for the GPIO pin and BLE. I/O handlers are described in Section IV.

This small example illustrates a key feature of Scoria needed by most IoT applications: concurrent programming with lightweight threads that can block on events. In C, it is easy to write a program that waits for an event from a single source by entering a polling loop, but waiting for multiple events is difficult. A central event-handler loop works but discards control state between events; the programmer must explicitly maintain any state, i.e., what to do next.

Multiple events are typically handled by OS-provided threads, which allows the programmer to write sequential code that only blocks on single events, but IoT devices often do not have the memory to provide a separate stack per thread. Protothreads [2] provides an extremely lightweight thread package in the form of C macros that only require storage for one additional program counter per thread, but the onus remains on the programmer to store all other state globally.

The Sparse Synchronous Model (SSM) of Edwards & Hui [3] integrates this blocking-thread concurrency model with a model of time that results in programs with deterministic functional and temporal behavior. A program running under SSM behaves like a discrete-event simulation: time is divided into instants in which the program executes. At any time, the program only knows what instant it is in and may schedule variable updates for a future instant. The SSM scheduler does its best to keep model time synchronized with real time, but the behaviour of the program is not affected by any deviation. Thread execution within a single instant is totally ordered to eliminate races and guarantee deterministic behavior.

Built atop SSM, Scoria is a programming language designed to simplify prototyping new language features and experiments with language design. Our compiler implements concurrency with C functions that store their state—control state and all local variables—in heap-resident activation records, instead of on the stack. Such code is like what a programmer might write with Protothreads, but Scoria provides more familiar abstractions such as local variables and blocking `wait` statements.

This paper describes the current status of Scoria and the choices we made during its development:

- We implement Scoria as an embedded domain-specific language (EDSL) in Haskell. It is mature enough that we can write programs that use I/O peripherals and run them on SoC-boards, but more work is needed before we can write complicated examples such as mesh networks.
- Programs in Scoria do not have to explicitly manage memory; the compiler and RTS allocate and deallocate memory, sparing us from a garbage collector.
- Scoria's language interface for interacting with I/O peripherals is the same as for interacting with regular Scoria references. Our RTS provides bindings for Zephyr OS [4].
- We test Scoria's compiler with QuickCheck [5], a property-based Haskell tester. We check for memory leaks and errors; compare the behaviour of compiled programs with an interpreter; and our domain-specific shrinker reduces errant programs a minimal test case.
- We evaluate the performance of Scoria's runtime system and generated code by stress testing it on real hardware. We precisely measure latencies to identify bottlenecks.

II. Motivation

To motivate real-time programming languages like Scoria, consider the "blinky" application commonly used as "hello world" in embedded systems programming. This toggles an LED at a specified frequency; Figure 3 shows the sample distributed with the Zephyr real-time operating systems (RTOS).

```
compiler :: Compile ()
compiler = do
  (gpio0, gpiohandler)  <- output 0
  (ble, _, scanhandler) <- enableBasicBLE
  let ?out0 = gpio0
      ?ble  = ble
  schedule scanhandler
  schedule entry
  schedule gpiohandler
```

Fig. 2: The top-level of the signal generator program in Figure 1. This requests a GPIO pin and a BLE radio handler, then runs the `entry` routine along with I/O handlers.

```
#include <zephyr/zephyr.h>
#include <zephyr/drivers/gpio.h>
static const struct gpio_dt_spec led =
    GPIO_DT_SPEC_GET(DT_ALIAS(led0), gpios);
void main(void) {
  int ret;
  if (!device_is_ready(led.port)) return;
  ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
  if (ret < 0) return;
  while (1) {
    ret = gpio_pin_toggle_dt(&led);
    if (ret < 0) return;
    k_msleep(1000); // sleep for 1000 ms
  }
}
```

Fig. 3: The "blinky" example distributed with Zephyr RTOS, which purports to generate a signal with a 2 s period.

While blinky (written in C) is adequate for showcasing the basic I/O and timing facilities of an embedded programming framework, it misrepresents its timing behavior: blinky does not actually blink once every two seconds! This discrepancy is due to two sources of latency: the program spends some time each loop iteration to toggle the LED, between invocations of `k_msleep` and once the timer set by `k_msleep` has expired, Zephyr takes some time to reschedule and resume the blinky program. If the time toggling the LED is Δ_l, and the time spent rescheduling the blinky program is Δ_s, then each loop iteration will take $1000 + \Delta_l + \Delta_s$ ms rather than just 1000 ms. Over time, the latency will accumulate into *drift* as the program continues to drag behind its ideal behavior (Figure 4).

To compensate for this latency and eliminate drift, the blinky program should only sleep for as long as it needs to keep up with the ideal timing behavior. To do this, our

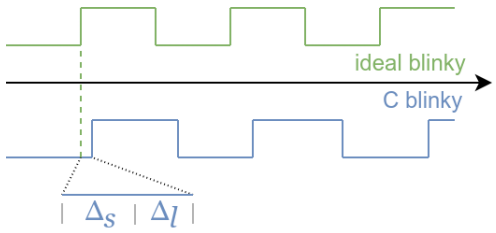

Fig. 4: The C blinky program of Figure 3 drifts due to latency from the RTOS (Δ_s) and the user program (Δ_l).

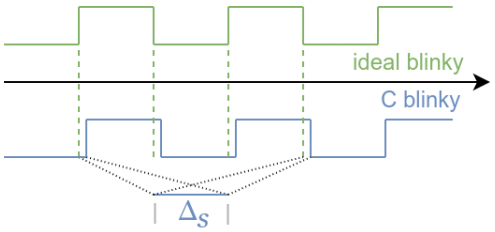

Fig. 5: Blinky without drift.

```
// include and defines
// declaration of button0, button1, led0 & led1

uint32_t last = -1;
void button_cb(const struct device *dev,
               struct gpio_callback *cb, uint32_t pins) {
        if (last == button0_pins && pins == button1_pins)
                gpio_pin_set_dt(&led0, 1);
        else if (last == button1_pins && pins == button0_pins)
                gpio_pin_set_dt(&led1, 1);
        else {
                gpio_pin_set_dt(&led0, 0);
                gpio_pin_set_dt(&led1, 0);
        }
        last = pins;
}

// callback initialization and registration
```

Fig. 6: While the program above is semantically simple, the implementation has to install interrupt handlers. Knowledge of which events have occurred must be lifted to the global scope, which must then be consulted before an LED can be lit.

"corrected" implementation (Figure A4) maintains a *logical* clock of how much time *should* have elapsed, and shortens its sleep time according to the difference between that logical time and the *actual* time. The resulting temporal behavior, shown in Figure 5, does not suffer from drift, only incurring a stable phase error of Δ_s.

However, this technique comes at the cost of complexity: the implementation shown in Figure A4 is 45 LOC, and is deeply entwined with Zephyr's API. Though certain configuration code may be simplified or omitted, the program's control flow and timing logic is obfuscated by the need to respond to the timer via interrupt handlers. Furthermore, this technique alone does not scale well beyond sequential programs, since any number of concurrent threads may need to synchronize their use of a finite number of hardware timers.

Scoria overcomes this complexity by incorporating the notion of logical time into its concurrent programming model. In doing so, Scoria allows the programmer to express the *intended* timing behavior, abstracting over platform-specific timer APIs. Scoria relies on its runtime to approximate the intended behavior to the extent possible by making use of available and supported timing capabilities. For comparison, Figure A1 shows blinky in Scoria, which is only 15 LOC.

Another example that illustrates how using interrupt handlers can complicate code unnecessarily is shown in Figure 6. It is an interrupt handler of a simple program that lights up one of two LEDs depending on which sequence of presses

```
seq :: Ref SW -> Ref SW -> Ref GPIO -> Ref GPIO -> SSM ()
seq b0 b1 l0 l1 = routine $ do
    wait (b0, b1)
    if changed b0 -- was b0 the button that was pressed?
        then do wait b1
                l0 <~ ON -- we know the sequence was b0, b1
        else do wait b0
                l1 <~ ON -- we know the sequence was b1, b0
```

Fig. 7: This Scoria routine implements the same behaviour as Figure 6, but each branch knows the source of its event and which LED to light without consulting a global "last" variable.

```
routine           -- Define a Scoria routine body
var       :: Exp a -> SSM (Ref a)      -- New local var
deref     :: Ref a -> Exp a            -- Access value
(<~)      :: Ref a -> Exp a -> SSM ()  -- Assignment
after     :: Time -> Ref a -> Exp a    -- Delayed
          -> SSM ()                    --   assignment
changed   :: Ref a -> Exp Bool         -- Was written?
wait      :: Waitable a => a -> SSM () -- Block on write
if-then-else :: Exp Bool -> SSM () -> SSM ()
          -> SSM ()                    -- Conditional
while     :: Exp Bool -> SSM () -> SSM () -- Loop
fork      :: [SSM ()] -> SSM ()        -- Concurrency
```

Fig. 8: The Core API of Scoria.

from two different buttons is registered. This program needs to remember which button was pressed previously, so that when the next press arrives it can inspect the complete sequence and decide which LED to light up. In order to remember the previous press, the program needs to use global variables, opening itself up to the possibility of complicated control flow where global variables are modified by several concurrently running threads. (Robert: this last part needs to be rephrased)

The Scoria version of this does not rely on global variables; it knows which events happened based on which code is executing and thus uses *local* information rather than global.

III. Overview of Scoria

Scoria is an embedded domain-specific language for IoT applications. It is domain-specific in that it is tailored to the demands of IoT applications through its facilities for concurrency and timing control. This makes the language easier to learn, compile, and optimize at the expense of making programming tasks outside its scope more difficult. It is embedded in a host language, Haskell, which allows us to benefit from the existing Haskell tools and compilers while presenting a convenient, new abstraction for the user. Haskell provides e.g the parser and a type checker, and we use host-level evaluation as a kind of macro system (see Section V). Such an approach enabled us to develop Scoria much more rapidly, making it easy to experiment with new language features.

We modeled Scoria on the proposal by Edwards & Hui [3], with adaptations to embed it in Haskell.

Figure 8 lists the core API of Scoria. These may be thought of as Scoria's primitives, but each are actually Haskell functions composed using Haskell's do-notation. Values that are embedded (Scoria values) are of type `Exp a`, while values

that belong to the host language are of type `a`. Host-level values are evaluated at Scoria compile-time.

The `routine` keyword introduces the body of a Scoria procedure. The Scoria compiler simply executes any Haskell code outside a `routine` block. See Section V for details.

A reference to a variable is created with the `var` keyword, can be assigned a new value with the `<~` ("assign") operator, and can be read with `deref`. The `changed` function returns true if a given variable (reference) has been written in the current instant, e.g., by an assignment in a concurrent thread.

The `after` operation schedules an assignment to a reference in a future instant. Its first argument is the time delay (relative to the current instant) before performing the assignment, and the other two arguments are the variable to update and the future value it should receive (which is evaluated immediately).

The `wait` statement blocks until at least one of the given variables (references) have been written, either by an immediate assignment `<~` or a scheduled assignment (`after`). `wait` takes either a single reference or tuples of references of varying sizes. Haskell's *type classes* [6] allow us to overload the operator to accept either.

`if-then-else` and `while` are the usual conditional and looping constructs. The code in their branches may block or call functions that do.

Lastly, `fork` is a parallel function call construct that spawns concurrently running child processes and blocks until all children have terminated (and not just blocked). The order of children is significant: it prescribes the order in which the children run in any instant, enforcing deterministic concurrency with a total execution order. In particular, an earlier child may write to a shared variable that a later child may read in the same instant, but not vice-versa (earlier children may only read data from a later child that was assigned in an earlier instant or in a scheduled update).

As shown in Figure 2, Scoria also provides a `Compile` monad that runs code at compile time. This is used to perform static initialization of e.g peripherals, and make them accessible to the rest of the Scoria program. As an example, a user might want to create a global variable `gv` and supply a reference to it to the `main` routine. This can be done as follows:

```
program :: Compile ()
program = do
  gv <- global @Word64
  let ?gv = gv
  schedule main

main :: (?gv :: Ref Word64) => SSM ()
main = routine $ ?gv <~ 5
```

To start the Scoria program, at least one routine must be added to the ready queue with the `schedule` directive, which populates the ready queue when the program begins. Each successive invocation of `schedule` adds another routine at a lower priority level: those scheduled first will run first in any instant, much like a `fork`.

a) Code generation from Scoria: Edwards & Hui [3] provide an SSM RTS coded in C. We have written a code generator that takes the abstract syntax of a Scoria program and produces C code that calls this RTS. The generated C code is platform agnostic; additional platform-specific code is necessary, e.g., to set a future wake-up alarm. We have written such bindings for Zephyr OS [4] and for something we call the trace platform, which we describe in Section VI.

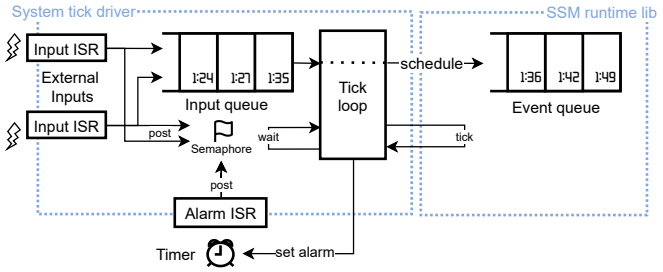

Fig. 9: The SSM runtime's input handling architecture.

IV. I/O in Scoria

A Scoria program communicates with its environment through variables. The Scoria runtime collects input events from interrupt service routines (ISRs)—effectively the callback functions of traditional asynchronous C implementations—and delivers them to the SSM event queue. Writes to output variables are observed by special I/O handler routines that typically call external C functions to perform the output.

a) Handling External Inputs and Time: Figure 9 shows the structure of the Scoria runtime that handles input events, manages the system timer, and runs the system in each instant. The main tick loop (Figure 11) retrieves events enqueued by ISRs (Figure 10), calls the SSM runtime's `tick()` of Edwards and Hui [3] to run the system (i.e., the code generated by the Scoria compiler) for a single instant, sets an alarm to wake it up at the next scheduled event, and then sleeps until it is awakened by the alarm or input from a peripheral.

Input events are gathered by ISRs and ultimately appear as scheduled variable updates to the Scoria program, but this process operates in several steps for safety and efficiency.

The Scoria runtime maintains two distinct event queues to avoid excessive synchronization costs. The input queue is loaded asynchronously by ISRs and emptied by the main tick loop, but it is a simple ring buffer that is easy to make thread-safe. By contrast, the main event queue is a priority queue to which events are added and sometimes removed out-of-order by both the tick loop and the various concurrent routines. However, because these run synchronously, the event queue is never accessed asynchronously and need not be thread-safe.

There are several reasons why we need two queues instead of just one. The synchronous hypothesis states that computations are instantaneous, meaning that an invocation of `tick` returns instantly. In reality this is not the case, and physical time will progress. If an external event is delivered to the system as soon as it arrives, `tick()` would look into the future, determinism would break, and we would be susceptible to data races. Furthermore, without time stamping external events, a second external event might arrive and overwrite the previous

```
void input_isr(device *dev) {
  int key = irq_lock();
  ssm_time_t input_time = timer_read();
  ssm_input_event *input = ssm_input_alloc();
  irq_unlock(key);
  if (input == NULL)  // No space in input queue
    return;
  input->time = input_time;
  input->data = ssm_input_read(dev); // Device-specific
  input->ref = ssm_input_binding(dev);
  ssm_input_commit(); // Enqueue input queue
  sem_post(ssm_sem);  // Wake up tick thread
}
```

Fig. 10: The structure of an input interrupt handler, which enqueues an event in the input queue. Each peripheral has its own device-specific handler.

```
struct semaphore ssm_sem;
void alarm_isr(void) { sem_post(&ssm_sem); }
void ssm_tick_loop(void) {
  ssm_init();
  ssm_tick();
  for (;;) {
    real_time = timer_read();
    model_time = ssm_next_time();
    ssm_input_event *input = ssm_input_peek();
    if (input && input->time <= model_time) {
      ssm_schedule_input(input); // Add to event queue
      ssm_input_release(); // Remove from input queue
    } else if (model_time <= real_time) {
      ssm_tick(); // Run system; advance model time
    } else if (model_time == NO_EVENT_SCHEDULED) {
      sem_wait(ssm_sem); // Wait for input
    } else {
      if (timer_set_alarm(model_time, alarm_isr))
        continue; // Alarm already expired
      sem_wait(ssm_sem); // Wait for alarm or input
      timer_cancel_alarm();
      sem_reset(ssm_sem);
    }
  }
}
```

Fig. 11: The Scoria runtime's main tick loop, which gathers events from the input queue, runs `tick()` to advance SSM model time, then sleeps until the next scheduled event or an interrupt.

event before the system has had a chance to observe the first one, leading to events essentially being dropped. The SSM runtime allows only one outstanding event per reference at a single time.

Figure 10 shows an input interrupt handler. It first records the current system time, then attempts to enqueue an event with that timestamp and any new data from the peripheral in the input queue: a ring buffer large enough to accommodate a modest backlog of input events before having to drop or overwrite events. Enqueue and dequeue operations are performed in-place via an allocate/commit and peek/release protocol to minimize copying.

In an ISR, recording the current system time as quickly as possible and allocating space in the input queue is critical to ensure enqueued input events appear with nondecreasing timestamps. Other higher-priority interrupts may occur while processing a lower-priority interrupt, but only after the timestamp of the lower priority interrupt has been captured.

Figure 11 shows the main tick loop, which waits on a OS-provided semaphore (`ssm_sem`) that is posted by an interrupt service routine from either a peripheral or the system clock. At each iteration of the tick loop, the runtime checks the input queue for events to schedule in the SSM event queue. Events in the input queue will be in increasing order since we assume the system timer advances monotonically. However, if the logical time of the SSM program is running behind physical time when some external input is received—`input->time` > `model_time`—or if there is no pending input event, the runtime executes the SSM program by calling `ssm_tick()`. Finally, if there are neither input events to process, nor internal events ready to execute, the main thread goes to sleep, blocking on a semaphore until either its timer expires or some fresh input appears.

b) Handling External Outputs: Just as with inputs, output peripherals are bound to regular Scoria references; writes to those references are emitted to the environment as output. Under the hood, those writes eventually trigger a call to some system-provided C output function that actualizes the output.

One of the design considerations we faced when implementing outputs was deciding exactly when in an instant the system output function should be invoked. The SSM model says all the code of an instant executes in zero time, but in practice any execution takes time and the code for an instant is effectively spread out over a finite time interval. If an output function is invoked too early, subsequent updates to the output variable in the same instant might be ignored, whereas if it is invoked too late, there will be output latency and perhaps even jitter arising from the execution speed of unrelated computation. For instance, consider the following three concurrent processes, shown in order of decreasing priority:

```
w1 :: Ref Button -> Ref LED -> SSM ()
w1 button led = routine $ do
  while true (do
    wait button
    led <~ true)

w2 :: Ref Button -> Ref LED -> SSM ()
w2 button led = routine $ do
  while true (do
    wait button
    led <~ false)

r :: Ref Button -> SSM ()
r button led = routine $ do
  while true (do
    wait button
    -- some expensive computation
  )
```

If the LED is lit immediately when it is first assigned—after the assignment of `true'` by `w1`—then the assignment of `false'` by `w2` will be missed, leading to a stale value being emitted. But if the LED is lit as late as possible—at the end of the instant—the system must wait until `r` has completed executing, incurring some delay. Depending on the particular application, either one of these outcomes may be unacceptable.

Our design leaves it to the user to decide when the physical output should take place by encapsulating it within an *output handler* process that the user must schedule. This handler is obtained when the user asks for the output device. If a process performs an immediate assignment to a reference, that assignment will only be visible to lower-priority processes. Note that delayed assignments do not have this behaviour; they always wake up every waiting process. For instance, if the user would like to schedule the handler after any possible assignments by `w1` and `w2`, but before `r`, the user may specify:

```
initProgram :: Compile ()
initProgram = do
  button <- input 0
  (led, handler) <- output 0

  schedule $ w1 button led
  schedule $ w2 button led
  schedule $ handler
  schedule $ r button led
```

While this precludes `r` from performing any effectful *instantaneous* assignments, it still accommodates effectful *delayed* assignments by `r`.

This gives the developer some control over two sources of jitter. The first is due to varying computation time—if an output handler is scheduled to run after a higher priority process in the same instant, then the actual output time will depend on the speed of the first process. This source of jitter can be eliminated by scheduling the output handler so it runs at the beginning of the instant at the expense of any instantaneous assignments—it is up to the user to balance the reliability versus the expressiveness of their application.

Discrepancies in input response times is a second source of jitter. The runtime system is able to handle inputs slightly faster when it is already awake than when it is asleep. If output is to be instantaneously triggered by some input, then the input-to-output latency will differ depending on whether an input is received when the system is asleep or awake.

V. Leveraging the Host Language

Scoria is implemented as a Haskell library that exposes an embedded language interface which allows us to reuse Haskell's parser and type checker, and otherwise take advantage of Haskell language features for rapid language development. In this section, we describe some of the embedding techniques we use to ensure that our language interface is robust, expressive, and easy to use.

a) Type Checking: Function Application in Haskell is checked to guarantee the type safety of the application. The value `routine body` has the same type as `body`, so Scoria procedures still look like ordinary Haskell functions and are invoked by performing normal Haskell function application, with the effect that the Haskell compiler will type check Scoria procedure invocations for us, for free. This kind of type checking happens in every corner of the language API, making it harder to construct illegal programs.

b) Providing New Abstractions: The host language of an embedded language can be used as a macro system, enabling effortless experimentation with language design. As an example, consider the `delay` statement, which blocks for a fixed period of time. This operation is not primitive to Scoria or SSM, but the behaviour can be recovered using existing primitives.

```
delay :: Exp Time -> SSM ()
delay x = do
  sync <- var event
  after x sync event
  wait sync
```

Because this Haskell function is not decorated with the `routine` keyword, it is not treated as a callable Scoria routine. Instead, when a program invokes `delay`, Scoria will inline `delay`'s body.

Yet this implementation has the downside of introducing a new local variable `sync` in the activation record of its call site each time `delay` is invoked. An alternative implementation which turns `delay` into a Scoria procedure might look like:

```
delay' :: Exp Time -> SSM ()
delay' x = fork [ delayRoutine x ]

delayRoutine :: Exp Time -> SSM ()
delayRoutine x = routine $ delay x
```

When a program invokes `delay'`, the inlined statement will instead be a `fork` statement that calls the auxiliary `delayRoutine` routine.

c) Plugin for Source-to-Source Transformations: Scoria uses the `routine` keyword to distinguish between a Haskell function and a Scoria procedure. This distinction is necessary because in the latter case, the compiler must construct AST nodes corresponding to the routine definition, rather than directly evaluating its body. It is this indirection that allows Scoria programs to perform recursion without endlessly inlining the routine body.

Yet routine definitions cannot be trivially embedded in Haskell, because it requires functions to reflect on their name and the names of their arguments. Case in point: an early iteration of our compiler used `make_routine` to construct routines, which expects additional arguments conveying the names of the routine and its arguments. This led to an awkward language interface with plenty of syntactic redundancy:

```
f = make_routine "f" ["x", "y"] $ \x y -> body
```

To overcome this burden, Scoria leverages plugin support from the Haskell compiler, GHC, to infer the arguments for `make_routine`. Our plugin looks for invocations of `routine` and replaces it with an invocation of `make_routine` with the necessary source information. For example, this source-to-source transformation produces the above boilerplate from the following:

```
f x y = routine body
```

VI. Testing

a) Interpretation: Following Edwards & Hui [3], we implemented an interpreter in Haskell that steps through

Bug location	Number of bugs
RTS	1
C generator	8
Interpreter	11

TABLE I: Breakdown of the bugs we identified developing of our compiler.

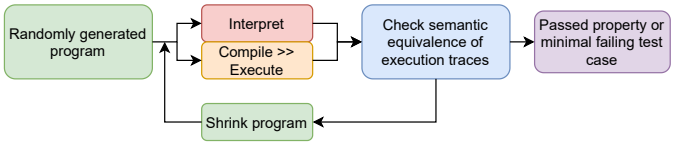

Fig. 12: Overview of how the second property is tested, asserting the semantic behaviour of SSM programs.

instants of model time but does not attempt to synchronize with wall-clock time. We use our interpreter as a reference to validate our compiler's output; see below.

b) Testing: To debug the Scoria compiler, we used QuickCheck [5] (QC), a property-based testing framework that generates *random* values and tests them against a property. If the property is falsified, QC shrinks the test to find a smaller failing case. The QC-generated random programs are typically too large to diagnose without shrinking.

We test two properties: the first is that generated C code compiles and runs without memory errors or leaks, according to Valgrind. This helped us find a bug related to our event queue: when a variable was deallocated, scheduled events were not being unscheduled and the RTS would attempt to perform updates using these stale pointers, corrupting the program state. We fixed this bug by ensuring that all routines unschedule outstanding updates to local variables before returning.

We also test semantic correctness of the generated program as shown in Figure 12. When testing, our compiler adds print statements to the generated code to produce an event trace, which we compare to one from the reference interpreter. This property helped us identify C generation problems. Most notably, we initially implemented Scoria signed integers with C's signed integers. Our tester found cases that branched on overflowing arithmetic to produce different behaviour because of the undefined overflow behavior of C's integers. We fixed this by switching to C's better-defined unsigned integers.

We documented 20 bugs, mostly semantic errors (Table I).

VII. Experimental Results

We test the performance of our Zephyr-based runtime system by subjecting Scoria programs to varying loads. All experiments are performed using a Nordic Semi NRF52840-DK board with a 64 MHz Cortex-M4 processor, 256 kB RAM, 1 MB flash, BLE support, and a 16 MHz crystal time base. We apply loads with a pulse generator driving a GPIO input pin and measure the results with an oscilloscope on an output pin.

a) Frequency Counter: To assess our implementation's resilience to high input load, we test a small Scoria frequency counter, shown in Figure 13. This program measures the frequency of button presses by counting the number of

```
freqCount :: Ref GPIO -> SSM ()
freqCount sw = routine $ do
  count <- var $ u32 0
  gate  <- var event
  after (secs 1) gate event
  while true (do
    if changed gate
      then do
        -- Print count to console
        if changed sw
          then count <~ 1
          else count <~ 0
        after (secs 1) gate event
        wait gate -- Sleep for 1 sec
        after (secs 1) gate event
      else
        count <~ deref count + 1
    wait (gate, sw))
freqCountProg :: Compile ()
freqCountProg = input 0 >>= schedule . freqCount
```

Fig. 13: A frequency counter program.

input events each second. The reported count is double the frequency, corresponding to the two input events of a single button press. For benchmarking purposes, we insert a print statement in the generated code to report the frequency; the program alternates between counting and reporting to ensure that the overhead of reporting the frequency does not interfere with the frequency counting at high loads.

The reference `sw` is bound to `input 0` on the NRF52840-DK; we connect a function generator to the corresponding GPIO pin and generate pulses at various frequencies. We found that our frequency counter is capable of measuring up events of up to 29 kHz (input frequency of 14.5 kHz, each rising or dropping edge creates one event), with error within 2 Hz. Beyond this, the input queue fills up faster than inputs are consumed, leading to events being dropped and the program thrashing. The frequency counter is able to recover from thrashing after we lower the input frequency below 14 kHz.

To measure the overhead of SSM event scheduling, we compare our Scoria frequency counter to the functionally equivalent manually coded C in Figure 14. The C implementation omits the SSM scheduler and directly manages synchronous control flow in `main_loop` using the same semaphore API provided by Zephyr OS. We also forego the input queue and directly increment `count` in the input ISR.

We find that this optimized version can accurately measure input frequencies of up to 34 kHz with 5 Hz of error, but is more verbose and error-prone, and less expressive and flexible.

b) Button-to-blink: To better understand the system sans synchronous logic, we first evaluate the "button-to-blink" program in Figure 15. This lights an LED when a button is pressed and turns it off when released. We schedule the LED handler after the main `b2b` program so that this instantaneous assignment is written to the LED at the end of the instant.

The instantaneous assignment in `b2b` incurs a practically avoidable but theoretically significant amount of latency, which we call the *at-rest input latency*, δ_r. This value represents the time it takes for the system to wake *from sleep*

```c
struct semaphore sem;
void alarm_isr(void) { sem_post(&sem); }

uint32_t count = 0;
// Executed each time button is pressed or released
void input_isr(void) { count++; }

void main_loop(void) {
  for (time_t time = timer_read(); true; count = 0) {
    timer_set_alarm((time += secs(1)), alarm_isr);
    sem_wait(&sem);
    printk("frequency␣*␣2:␣%u␣Hz\r\n", count);
    timer_set_alarm((time += secs(1)), alarm_isr);
    sem_wait(&sem);
  }
}
```

Fig. 14: Frequency counter program implemented in C.

```
b2b :: (?sw :: Ref Switch, ?led :: Ref GPIO) => SSM ()
b2b = routine $ while true $ do
    wait ?sw
    ?led <~ deref ?sw

b2bc :: Compile ()
b2bc = do
    sw <- input 0
    (led, handler) <- output 0
    let ?sw = sw
        ?led = led
    schedule b2b
    schedule handler
```

Fig. 15: Button-to-blink in Scoria; generated C in Figure A11

and respond to external input. From profiling, we find that δ_r for our NRF52840-DK is approximately 60 µs.

When the system receives inputs at a frequency beyond $1/\delta_r$, the system will not be able to process an input before receiving the next, making δ_r a significant number. As the system falls farther and farther behind wall clock time, its behaviour will degrade into an asynchronous system with none of the temporal behaviour of the underlying Scoria program.

A related metric is the *in-flight input latency*, δ_f: the time it takes for a busy system to respond to external input. When events arrive separated by an amount of time within δ_f, the input queue will be populated with new events by the time the main tick loop thread checks it again, meaning it can resume ticking without putting itself to sleep. δ_f is shorter than δ_r because it eliminates the time spent sleeping and waking. We measured δ_f on our board to be about 45 µs.

Sparse bursts of events do not overload the system if it can catch up between bursts. The system may even be able to keep up if it is consistently stimulated with a period between δ_r and

Frequency (kHz)	Latency (µs)	Jitter (µs)
2	60.0	0.7
4	60.6	0.7
6	49.7	0.5
8	52.1	13.1
10	47.3	34.0
12	45.6	24.0

TABLE II: Latency and jitter of button-to-blink (Figure 15)

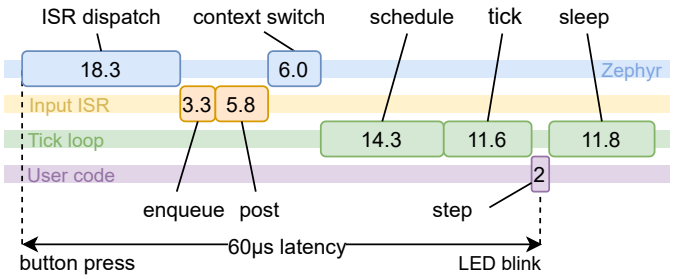

Fig. 16: Timeline in µs of a single button press of "button-to-blink" (Figure 15), reconstructed from GPIO profiling.

δ_f. As shown in Table II, the "button-to-blink" program is able to sustain activity when stimulated by a pulse generator with frequencies of 12 kHz; beyond 12 kHz, the program begins to thrash and drop input events. However, when the delay between successive events is less than δ_r, the computation time per instant becomes less predictable, even as the system remains responsive. In Table II, this manifests in increased jitter at input frequencies beyond 8 kHz, at which a square wave has a half-period of 62.5 µs.

To determine the source of our system's δ_r of 60 µs, we profile a single button press of b2b to obtain the timeline in Figure 16. We add code that emits 4-bit codes on GPIO pins for certain events and record them with a logic analyzer. Analyzing the recorded data determines time between events. While this approach is intrusive, we found each GPIO write took only 60 ns, negligible compared to what we are measuring. We find that 24.3 µs is directly introduced by Zephyr's own ISR and process scheduling facilities. A further 5.8 µs is introduced by Zephyr's semaphore implementation, when we call `sem_post` from the ISR to wake up the tick loop thread. The δ_r of 60 µs is from Table II; Figure 16 shows slightly different numbers from a different Zephyr version.

We also measure δ_r for two Zephyr programs written in C to compare with our Scoria implementation. One of our implementations, Figure A2, unrealistically constantly polls for button presses, and represents the lowest theoretical at-rest response time for any Zephyr program. Our other implementation is more realistic, responding to input via an interrupt handler that reports the button press event by placing it in a ring buffer identical to the one in the Scoria runtime and resuming the main thread by releasing a semaphore. The main thread then unblocks, retrieves the event from the ring buffer, and updates the LED. Figure A3 shows this version, which models the Scoria runtime without the SSM "tick" scheduler. We standardized the input event frequency at 2 kHz across all implementations; our results are shown in Table III.

The Scoria version is the slowest of the three. Figure 16 shows the amount of time spent by Zephyr locating and running the input ISR was roughly 33 µs. This is comparable to the response time measured by the more realistic Zephyr example (Table III). The overhead imposed by the Scoria runtime system is mainly the parts in green (the *Tick loop* bar), where the Scoria runtime is retrieving the event from

Program	Mean δ_r (µs)	Min. δ_r (µs)	Max. δ_r (µs)
Polling	1.519	0.88	2.17
Interrupt + sem	31.46	31.32	31.59
Scoria	57.96	57.86	58.07

TABLE III: The at-rest response time of three button-to-blink implementations fed a 2 kHz input. Data gathered with a logic analyzer sampling at 100 MHz.

Program	Generated Frequency (kHz)	LOC
C	12.2	54
Scoria	6.57	32

TABLE IV: Implementations' maximum frequencies and sizes

the input queue, inserting it in the Scoria environment, and scheduling processes to invoke the Scoria tick loop.

c) Signal Generator: With these measurements made we can return to the signal generator from Figure 1. We measure the highest frequency (shortest period) it can generate. For the purposes of evaluation, we adapted the example to adjust the half-period `hperiod` in linear increments of 2 µs by the press of a button. Figure A10 shows the complete code. We find that the signal generator from Figure 1 can reliably generate signals with half-period as low as 76 µs, corresponding to a frequency of about 6.6 kHz, with less than 500 ns of jitter—within the level of precision of our oscilloscope. At half-periods between 70 µs–76 µs, we observe degradation in signal accuracy and consistency: the output signal's half-period fluctuates between 60.2 µs–83.4 µs. At lower half-periods, the system ceases to output any signal as it is overwhelmed by the computation load. Once the half-period is increased beyond 76 µs, the system eventually recovers after a burst of high-frequency output, emitted as it catches up with wall-clock time.

Table IV compares our Scoria implementation with a comparable signal generator written in C (Figure A9). We measure the maximum frequency for each implementation. Like the Scoria runtime, the C implementation populates a ring buffer when the alarm goes off; the main thread is then unblocked by a semaphore release, after which it toggles the LED.

The Zephyr program outperforms the Scoria program, as can be seen in table IV. As soon as the half period becomes lower than the time it takes for the program to handle one wave event, the signal becomes unstable. Since the Scoria runtime adds some overhead when reacting to events, as can be seen in Figure 16, the Scoria program can not handle as low half periods as the Zephyr program. The Zephyr program can generate about twice as high of a frequency.

d) BLE Frequency Mime: As a slightly more complicated example, we wrote an application where two boards communicate over BLE. One board will advertise varying frequencies using BLE, and the other board will scan for such messages. When a message is intercepted, the board will generate a signal of the detected frequency as it continues to scan for messages.

The C version needs to manage such things as callbacks, clocks, counters, and global state. Any of these areas could hide bugs, and they are all absent from the Scoria version. The Scoria version specifies only application logic, not the details of how it is realized. Figures A5 to A8 show both versions.

VIII. Related Work

a) Synchronous Languages: Scoria's programming model SSM [3] adopts the *synchrony hypothesis*: concurrent execution proceeds as a sequence of synchronized, zero-time instants [7]. While digital hardware designers have used this model for over half a century, French researchers in the 1990s brought the concept to software languages Lustre, Esterel, and Signal [8]–[10]. Synchrony simplifies reasoning about concurrent timing behavior by eliminating the non-determinism of asynchronony, and has since featured in languages such as Timber [11] and Lingua Franca [12], and EDSLs like Haski [13] and Copilot [14].

Despite their application in real-time programming, the "French school" of programming languages—Lustre, Esterel, and Signal [8]–[10]—do not include time as a first-class language construct. Instead, they specify computation through *successive* instants, and compile to tick functions that must be invoked periodically by a cyclic executive for every instant, even those where no computation takes place. Meanwhile, Scoria's `after` statement allows users to specify (or compute) concrete time delays, and its sparse execution model does not obligate its runtime to compute inert instants.

Many of the synchronous languages are dataflow-oriented, including Lustre, Signal, and Lingua Franca [8], [10], [12], where programs specify a static, finite dataflow graph where processes (nodes) communicate along fixed channels (edges). In contrast, Scoria is imperative (programs specify control flow); processes are spawned dynamically using `fork`, and the dataflow between them is implicit. Though Esterel is also imperative and includes concurrent control-flow constructs like Scoria's `fork` and `wait`, it lacks function calls, limiting Esterel programs to a finite number of processes [9].

Like Scoria, Haski [13] and Copilot [14] are deeply embedded Haskell DSLs [15] that generate C code for microcontrollers. Haski is an embedding of Lustre in Haskell for secure IoT programming; Copilot is an EDSL for monitoring software that promises constant-time and -space C code. Both are dataflow languages, so their embedding in Haskell constructs abstract streams instead of Scoria's syntax trees.

b) Compiler Testing: While manually written regression test suites remain the standard for compiler development, many have also considered randomly generating and shrinking compiler test cases [16], [17]. Our work is most similar to RandIR [18], which uses ScalaCheck [19] to test LMS [20], a code generation framework for embedded DSLs. RandIR uses Scala as its test oracle, translating its test candidates to Scala to compare against other language backends; in comparison, Scoria's test oracle directly interprets generated Scoria IR.

For well-established languages, differential testing can take advantage of multiple compiler implementations, e.g., CSmith for C [21], or multiple optimization levels, e.g., Palka et al. [22] for the Glasgow Haskell Compiler. While Scoria cannot afford such an opportunity as a nascent language, we look forward to adapting our test-driven compiler development to help identify bugs as we begin to implement optimizations that take advantage of Scoria's synchronous semantics.

c) Discrete-Event Systems: Ptides [23] is a programming model used to implement discrete-event real-time systems that inspired SSM and Scoria. Unlike Scoria/SSM, however, Ptides implements a *distributed* discrete-event system that cautiously determines when it is safe to process a timestamped message, i.e., when a message with an earlier timestamp cannot appear. We plan to implement distributed Scoria following Ptides.

IX. Future Work

a) Types: Scoria currently only supports a limited, predefined set of data types. We will add support for more complex types, such as arrays and user-defined algebraic data types (ADTs). Building on McDonnell et al. [24], we can expose a pattern-matching language interface that is familiar to Haskell programmers. We believe ADTs will be a good fit for writing IoT programs; the richness they bring will be especially helpful for developing safe interfaces for peripherals.

b) Linear Types: Linear types [25] were recently added to GHC, which allows certain invariants to be encoded in the types and checked by the compiler. For instance, the compiler can enforce that we schedule our I/O handlers, if `output` and `schedule` are given more precise linear types:

```
output :: Int
        -> (Ref LED -> SSM () %1 -> Compile ())
        -> Compile ()
output i continuation = ...

schedule :: SSM () %1 -> Compile ()
schedule handler = ...
```

`output`'s type indicates that it takes an integer identifying the LED, as well as a *continuation* that takes the LED reference and handler as parameters, where the `%1` indicates that handler must be consumed exactly once. These functions can be invoked as follows:

```
program :: Compile ()
program = output 0 $ \ref handler -> do
  schedule handler -- followed by rest of program setup
```

Since the handler is scheduled inside of the continuation, forgetting to schedule the handler (or scheduling it more than once) is a GHC compile time error. This could otherwise only be checked at code-generation time (host-language run time).

c) Performance: Experimentally, we identified a few sources of inefficiency that make Scoria significantly slower than counterparts implemented directly in C and Zephyr. While the C and Zephyr versions process an event as soon as the associated interrupt handler is invoked, Scoria has to enqueue the message and wake up the main tick-thread that retrieves it and actually runs the program, which add overhead.

The benchmarked programs can perform significantly better if the required OS facilities were developed directly on the bare metal rather than acquired by Zephyr. While Zephyr gives quick access to many convenient abstractions, they are developed in a very general way to be suitable for many kind of applications. If we could tailor these abstractions to just the needs of Scoria, we might be able to get some speedup. We will try this, as well as trying out other operating systems.

X. Conclusions

While the Scoria language is still a simple language, Scoria programs are short and clutter-free, and the compiler generates resource-efficient code. Implementing Scoria as an EDSL has led to several advantages for language and compiler development. Taking advantage of features and tooling provided by the host language, we can rapidly prototype our language. Using primitives to derive new operations such as `delay` is straightforward and transparent. Since the EDSL itself is simply a Haskell library whose API constructs an AST, we can easily experiment with exposing and modifying compiler internals with little coordination with the language front-end.

We were able to confidently iterate on our compiler due to our Scoria interpreter, which we developed early on. Using the interpreter as a language oracle, we used QuickCheck to thoroughly and efficiently test corner cases in our language, identifying several bugs that would have otherwise taken much longer to identify. To help us better understand what went wrong, our shrinker produced minimal failing test programs which easily pointed to the source of each bug we found.

We extended the RTS proposed by Edwards & Hui [3] with Zephyr OS bindings to run Scoria programs on real hardware. I/O peripherals are exposed via Scoria references so that Scoria programs can converse with peripherals natively, avoiding the callback hell typical of IoT applications. This interaction model has allowed us to easily add support for a handful of peripherals, including GPIO and BLE. Scoria allows users to precisely specify the priority of I/O handling code relative to concurrent user code and make application-specific reactivity and sensitivity trade-offs. Our stress tests confirm we can write real-time applications with robust overflow behaviour.

While Scoria remains an experimental language, our successes with it to date have illustrated how implementing an EDSL provides a faster route to language implementation for experimental languages. As expected, Scoria has raised many questions about how any sparse synchronous language should be designed, notably how its interface with the outside world should be specified, and given us a quick route to experimenting with solutions to these problems. Work on Scoria continues and will, at the very least, inform language design in similar domains.

Acknowledgements

Hui and Edwards were supported in part by the National Institutes of Health (NIH) under grant 1RF1MH120034-01.

Krook and Claessen were supported by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Swedish research agency Vetenskapsrådet under the project SyTeC (Ref. 2016-06204).

An additional thanks goes to Agustín Mista for help writing the plugin described in this paper.

References

[1] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4es, pp. 196–es, dec 1996. [Online]. Available: https://doi.org/10.1145/242224.242477

[2] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 29–42. [Online]. Available: https://doi.org/10.1145/1182807.1182811

[3] S. A. Edwards and J. Hui, "The sparse synchronous model," in *Forum on Specification and Design Languages (FDL)*. Kiel, Germany: IEEE, Sep. 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FDL50818.2020.9232938

[4] The Zephyr Project, "The Zephyr Project," https://www.zephyrproject.org/, 2021.

[5] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, vol. 35, no. 9. New York, NY, USA: Association for Computing Machinery, Sep. 2000, pp. 268–279. [Online]. Available: https://doi.org/10.1145/357766.351266

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler, "Type classes in haskell," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 2, pp. 109–138, 1996.

[7] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, jan 2003.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 178–188. [Online]. Available: https://doi.org/10.1145/41625.41641

[9] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[10] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the signal language and its semantics," *Science of computer programming*, vol. 16, no. 2, pp. 103–149, 1991.

[11] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *Asian Symposium on Programming Languages and Systems*, 2003, pp. 339–356. [Online]. Available: https://doi.org/10.1007%2F978-3-540-40018-9_22

[12] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 4, pp. 1–27, Jul. 2021. [Online]. Available: https://doi.org/10.1145%2F3448128

[13] N. Valliappan, R. Krook, A. Russo, and K. Claessen, "Towards secure iot programming in haskell," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 136–150. [Online]. Available: https://doi.org/10.1145/3406088.3409027

[14] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *International Conference on Runtime Verification*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 345–359.

[15] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding for edsl," in *Trends in Functional Programming*, H.-W. Loidl and R. Peña, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–36.

[16] A. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *Information and Software Technology*, vol. 39, no. 9, pp. 617–625, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584997000177

[17] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, may 2020. [Online]. Available: https://doi.org/10.1145%2F3363562

[18] G. Ofenbeck, T. Rompf, and M. Püschel, "RandIR: differential testing for embedded compilers," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM, oct 2016. [Online]. Available: https://doi.org/10.1145%2F2998392.2998397

[19] R. Nilsson, *ScalaCheck: The Definitive Guide*. Artima Press, 2014. [Online]. Available: https://books.google.se/books?id=AOvcoQEACAAJ

[20] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," *Communications of the ACM*, vol. 55, no. 6, pp. 121–130, jun 2012. [Online]. Available: https://doi.org/10.1145%2F2184319.2184345

[21] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[22] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 91–97. [Online]. Available: https://doi.org/10.1145/1982595.1982615

[23] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zheo, and J. Zou, "Ptides: A programming model for distributed real-time embedded systems," CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, Tech. Rep., 2008.

[24] T. L. McDonell, J. D. Meredith, and G. Keller, "Embedded pattern matching," *arXiv preprint arXiv:2108.13114*, 2021.

[25] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. P. Jones, and A. Spiwack, "Linear haskell: practical linearity in a higher-order polymorphic language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, jan 2018. [Online]. Available: https://doi.org/10.1145%2F3158093

APPENDIX

```haskell
{-# LANGUAGE ImplicitParams #-}
{-# OPTIONS_GHC -fplugin=SSM.Plugin
                -fplugin-opt=SSM.Plugin:mode=routine #-}
module Blinky where

import SSM.Language
import SSM.Frontend.Peripheral.LED

program :: (?pin :: Ref GPIO) => SSM ()
program = routine $ while true $ do
  after (msecs 1000) ?pin (not $ deref ?pin)
  wait ?pin

main :: Compile ()
main = do
  (pin, handler) <- output 1
  let ?pin = pin
  schedule handler
  schedule program
```

Fig. A1: The "blinky" example from Figure A4, implemented in Scoria.

```c
#include <zephyr.h>
#include <drivers/gpio.h>

static const struct gpio_dt_spec button =
    GPIO_DT_SPEC_GET_OR(DT_ALIAS(sw0), gpios, {0});

static struct gpio_dt_spec led =
    GPIO_DT_SPEC_GET_OR(DT_ALIAS(led0), gpios, {0});

void main() {
    gpio_pin_configure_dt(&button, GPIO_INPUT);
    gpio_pin_configure_dt(&led, GPIO_OUTPUT);

    while (1)
        gpio_pin_set_dt(&led, gpio_pin_get_dt(&button));
}
```

Fig. A2: The code of the polling button-to-blink code. 100% of the CPU is allocated to reflecting the button state on the LED, which gives a kind of theoretical minimum for how fast the response time *could* be.

```c
#include <zephyr.h>
#include <drivers/gpio.h>
#include "rb.h"

K_SEM_DEFINE(my_sem, 0, 1);
SSM_RB_DEFINE(int, ssm_input_buffer, 12);

static const struct gpio_dt_spec button =
    GPIO_DT_SPEC_GET_OR(DT_ALIAS(sw0), gpios, {0});
static struct gpio_dt_spec led =
    GPIO_DT_SPEC_GET_OR(DT_ALIAS(led0), gpios, {0});
static struct gpio_callback button_cb_data;

void button_pressed(const struct device *dev
                    , struct gpio_callback *cb
                    , uint32_t pins) {
    int *msg = ssm_rb_writer_alloc(ssm_input_buffer);
    if(msg) {
        *msg = gpio_pin_get_dt(&button);
        ssm_rb_writer_commit(ssm_input_buffer);
    }
    k_sem_give(&my_sem);
}

void main(void)
{
    gpio_pin_configure_dt(&button, GPIO_INPUT);
    gpio_pin_interrupt_configure_dt(&button,

    gpio_init_callback(&button_cb_data
                       , button_pressed
                       , BIT(button.pin));
    gpio_add_callback(button.port, &button_cb_data);

    gpio_pin_configure_dt(&led, GPIO_OUTPUT);

    while(true) {
        k_sem_take(&my_sem, K_FOREVER);
        int *msg = ssm_rb_reader_claim(ssm_input_buffer);
        if(msg) {
            gpio_pin_set_dt(&led, *msg);
            ssm_rb_reader_free(ssm_input_buffer);
        }
    }
}
```

Fig. A3: The button-to-blink program implemented using Zephyr, where input events are delivered to an input ISR (line 14). When an event arrives, it is placed in a ring buffer (line 17-20), and the main thread is woken up by releasing a semaphore (line 22). The main thread retrieves the event from the ring buffer (line 39-40) and toggles the LED.

```c
#include <zephyr.h>
#include <drivers/gpio.h>
#include <drivers/counter.h>
#include <drivers/clock_control.h>
#include <drivers/clock_control/nrf_clock_control.h>

static struct gpio_dt_spec led = GPIO_DT_SPEC_GET_OR(DT_ALIAS(ssm_led), gpios, {0});
const struct device *timer_dev, *clock_dev;
struct counter_alarm_cfg alarm_cfg;
uint32_t current; // Logical time

// Make sure GPIO device for LED is ready, and configure it
int configure_led(void) {
  if (led.port && !device_is_ready(led.port))
    return -ENODEV;
  gpio_pin_configure(&led, GPIO_OUTPUT);
}

// Configure timer device to use NRF52840-DK's external crystal oscillator
int configure_timer(void) {
  timer_dev = device_get_binding(DT_LABEL(DT_ALIAS(ssm_timer)));
  if (!timer_dev)
    return -ENODEV;
  counter_start(timer_dev);

  clock_dev = device_get_binding(DT_LABEL(DT_INST(0, nordic_nrf_clock)));
  if (!clock_dev)
    return -ENODEV;
  clock_control_on(clock_dev, CLOCK_CONTROL_NRF_SUBSYS_HF);
}

// Set an alarm to go off in delay_in_us microseconds, relative to logical time
void set_alarm(uint64_t delay_in_us) {
  alarm_cfg.ticks = (current += counter_us_to_ticks(timer_dev, delay_in_us));
  counter_set_channel_alarm(timer_dev, 0, &alarm_cfg);
}

// Called when the alarm goes off
void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id,
                          uint32_t ticks, void *user_data) {
  gpio_pin_toggle_dt(&led);
  set_alarm(1000000); // 1000000 us = 1000 ms
}

// Configure alarm to call counter_interrupt when wakeup time is reached
int configure_alarm(void) {
  alarm_cfg.flags = COUNTER_ALARM_CFG_ABSOLUTE | COUNTER_ALARM_CFG_EXPIRE_WHEN_LATE;
  alarm_cfg.callback = counter_interrupt_fn;
  alarm_cfg.user_data = &alarm_cfg;

  // Initialize current logical time
  counter_get_value(timer_dev, &current);
}

void main(void) {
  configure_led();
  configure_timer();
  configure_alarm();
  set_alarm(1000000); // 1000000 us = 1000 ms
}
```

Fig. A4: "Blinky" implementation with correct temporal behavior, implemented using Zephyr RTOS. To achieve more precise timing, the code configures the NRF52840-DK board to use its external crystal oscillator as its clock source. This implementation does not suffer from drift due to its use of logical time, though this comes at the complexity of (1) maintaining the value of the logical clock, and (2) decomposing the application into interrupt handlers.

```c
#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>

#include <drivers/counter.h>
#include <drivers/clock_control.h>
#include <drivers/clock_control/nrf_clock_control.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>

static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data);

int ssm_timer_configure_clock(void) {
    const struct device *clock;
    clock = device_get_binding(DT_LABEL(DT_INST(0, nordic_nrf_clock)));
    clock_control_on(clock, CLOCK_CONTROL_NRF_SUBSYS_HF);
    return 0;
}

const struct device *timer_dev;
struct counter_alarm_cfg alarm_cfg;

void set_alarm(uint64_t delay_in_us) {
    alarm_cfg.ticks = counter_us_to_ticks(timer_dev, delay_in_us);
    counter_set_channel_alarm(timer_dev, 0, &alarm_cfg);
}

uint64_t broadcast_data = 0;
static struct bt_data ad[] = {
    BT_DATA(BT_DATA_MANUFACTURER_DATA, &broadcast_data, 8)};

K_THREAD_STACK_DEFINE(my_stack_area, 512);

struct k_work_q my_work_q;

#define SHORT_FREQ 200000
#define LONG_FREQ  500000

void alternate_frequency_broadcast(struct k_work *item) {
    bt_le_adv_stop();
    broadcast_data = broadcast_data == SHORT_FREQ ? LONG_FREQ : SHORT_FREQ;
    bt_le_adv_start(BT_LE_ADV_NCONN_IDENTITY, ad, ARRAY_SIZE(ad), NULL, 0);
}

K_WORK_DEFINE(my_work_item, alternate_frequency_broadcast);

static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data) {
    k_work_submit(&my_work_item);
    set_alarm(5000000);
}

void main() {
    bt_enable(NULL);

    k_work_queue_start(&my_work_q, my_stack_area, K_THREAD_STACK_SIZEOF(my_stack_area), 5, NULL);

    timer_dev = device_get_binding(DT_LABEL(DT_ALIAS(ssm_timer)));
    counter_start(timer_dev);
    ssm_timer_configure_clock();

    alarm_cfg.flags = 0;
    alarm_cfg.callback = counter_interrupt_fn;
    alarm_cfg.user_data = &alarm_cfg;

    set_alarm(5000000);
}
```

Fig. A5: The C version of the BLE mime broadcaster. This application broadcasts one of two frequencies, switching between the two every five seconds.

```c
#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>

#include <drivers/counter.h>
#include <drivers/clock_control.h>
#include <drivers/clock_control/nrf_clock_control.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>

static struct gpio_dt_spec led = GPIO_DT_SPEC_GET_OR(DT_ALIAS(led0), gpios, {0});

static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data);

int ssm_timer_configure_clock(void) {
    const struct device *clock;
    clock = device_get_binding(DT_LABEL(DT_INST(0, nordic_nrf_clock)));
    clock_control_on(clock, CLOCK_CONTROL_NRF_SUBSYS_HF);
    return 0;
}

const struct device *timer_dev;
struct counter_alarm_cfg alarm_cfg;

void set_alarm(uint64_t delay_in_us) {
    alarm_cfg.ticks = counter_us_to_ticks(timer_dev, delay_in_us);
    counter_set_channel_alarm(timer_dev, 0, &alarm_cfg);
}

uint64_t period_in_us = 1000000;
uint64_t next_value = 0;

static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data) {
    gpio_pin_set_dt(&led, next_value);
    next_value = next_value ^ 1UL;
    set_alarm(period_in_us);
}

static void device_found(const bt_addr_le_t *addr, int8_t rssi, uint8_t type, struct net_buf_simple *ad) {
    if (type != BT_GAP_ADV_TYPE_ADV_NONCONN_IND)
        return;

    if (ad->data[0] == 9 && ad->data[1] == BT_DATA_MANUFACTURER_DATA) {
        uint64_t res = 0;
        for (int i = 7; i >= 0; i--) {
            res = (res << 8) | (uint64_t)ad->data[2 + i];
        }
        period_in_us = res;
    }
}

void main() {
    bt_enable(NULL);

    timer_dev = device_get_binding(DT_LABEL(DT_ALIAS(ssm_timer)));
    counter_start(timer_dev);
    ssm_timer_configure_clock();

    alarm_cfg.flags = 0;
    alarm_cfg.callback = counter_interrupt_fn;
    alarm_cfg.user_data = &alarm_cfg;

    bt_le_scan_start(BT_LE_SCAN_PASSIVE, device_found);
    gpio_pin_configure_dt(&led, GPIO_OUTPUT);
    set_alarm(period_in_us);
}
```

Fig. A6: The C version of the BLE mime generator. This application scans for broadcasted messages, and generates a frequency on LED 0. If a broadcasted message is detected, the generated frequency is the one specified in the advertisement payload.

```
{-# LANGUAGE ImplicitParams #-}
{-# OPTIONS_GHC -fplugin=SSM.Plugin -fplugin-opt=SSM.Plugin:mode=routine #-}
module FrequencyMime where

import SSM.Language
import SSM.Frontend.Peripheral.BasicBLE

freqCount :: Ref Time -> SSM () -- concurrent process dictating which frequency to broadcast
freqCount period = routine $ while true $ do
    period <~ (msecs 200)
    delay (secs 5)
    period <~ (msecs 500)
    delay (secs 5)

broadcastCount :: (?ble :: BBLE) => Ref Time -> SSM ()
broadcastCount count = routine $ while true $ do
    enableBroadcast $ time2ns $ deref count
    delay (secs 5)
    disableBroadcast

counterEntry :: (?ble :: BBLE) => SSM ()
counterEntry = routine $ do
    count <- var $ secs 1 -- create the reference that is shared between the concurrent processes
    fork [ freqCount count, broadcastCount count ]

counter :: Compile ()
counter = do
    (ble, broadcast, scanning) <- enableBasicBLE

    let ?ble = ble

    schedule counterEntry
    schedule broadcast
    schedule scanning
```

Fig. A7: The Scoria version of the BLE mime broadcaster. It implements the same application as the one in Figure A5. See Section V for a description of what `delay` does.

```
{-# LANGUAGE ImplicitParams #-}
{-# OPTIONS_GHC -fplugin=SSM.Plugin -fplugin-opt=SSM.Plugin:mode=routine #-}
module FrequencyMime where

import SSM.Language
import SSM.Frontend.Peripheral.LED
import SSM.Frontend.Peripheral.BasicBLE

bleHandler :: (?ble :: BBLE) => Ref Time -> SSM () -- scan for broadcasted frequencies and write them to the shared reference
bleHandler period = routine $ do
    enableScan
    while true $ do
      wait scanref
      period <~ nsecs (deref scanref)
      delay (secs 5)

freqGen :: (?led0::Ref LED) => Ref Time -> SSM () -- generate a frequency with a half period dictated by the value of the shared reference
freqGen period = routine $ while true $ do
  after (deref period) ?led0 (not' $ deref ?led0)
  wait ?led0

entry :: (?ble :: BBLE, ?led0::Ref LED) => SSM ()
entry = routine $ do
  period <- var $ secs 1 -- create reference that is shared between the concurrent processes
  fork [freqGen period, bleHandler period]

generator :: Compile ()
generator = do
  (led, handler) <- output 0
  (ble, broadcast, scanning) <- enableBasicBLE

  let ?led0 = led
      ?ble  = ble

  schedule handler
  schedule entry
  schedule broadcast
  schedule scanning
```

Fig. A8: The Scoria version of the BLE mime generator. It implements the same application as the one in Figure A6. See Section V for a description of what `delay` does.

```
1
2  #include <zephyr.h>
3  #include <drivers/gpio.h>
4
5  #include <drivers/counter.h>
6  #include <drivers/clock_control.h>
7  #include <drivers/clock_control/nrf_clock_control.h>
8
9  #include "rb.h"
10
11 K_SEM_DEFINE(my_sem, 0, 1);
12 SSM_RB_DEFINE(uint8_t, ssm_input_buffer, 12);
13
14 static struct gpio_dt_spec led = GPIO_DT_SPEC_GET_OR(DT_ALIAS(led0), gpios, {0});
15
16 static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data);
17
18 uint32_t current;
19 const struct device *timer_dev;
20 struct counter_alarm_cfg alarm_cfg;
21
22 int ssm_timer_configure_clock(void) {
23     const struct device *clock;
24     clock = device_get_binding(DT_LABEL(DT_INST(0, nordic_nrf_clock)));
25     clock_control_on(clock, CLOCK_CONTROL_NRF_SUBSYS_HF);
26     return 0;
27 }
28
29 void set_alarm(uint64_t delay_in_us) {
30     alarm_cfg.ticks = (current += counter_us_to_ticks(timer_dev, delay_in_us));
31     counter_set_channel_alarm(timer_dev, 0, &alarm_cfg);
32 }
33
34 uint8_t next;
35
36 static void counter_interrupt_fn(const struct device *counter_dev, uint8_t chan_id, uint32_t ticks, void *user_data) {
37     next = next ? 0 : 1;
38     uint8_t *msg = ssm_rb_writer_alloc(ssm_input_buffer);
39     if(msg) {
40         *msg = next;
41         ssm_rb_writer_commit(ssm_input_buffer);
42     }
43     k_sem_give(&my_sem);
44 }
45
46 void main() {
47     gpio_pin_configure_dt(&led, GPIO_OUTPUT);
48
49     timer_dev = device_get_binding(DT_LABEL(DT_ALIAS(ssm_timer)));
50     counter_start(timer_dev);
51     ssm_timer_configure_clock();
52     counter_get_value(timer_dev, &current);
53
54     alarm_cfg.flags = COUNTER_ALARM_CFG_ABSOLUTE | COUNTER_ALARM_CFG_EXPIRE_WHEN_LATE;
55     alarm_cfg.callback = counter_interrupt_fn;
56     alarm_cfg.user_data = &alarm_cfg;
57
58     uint32_t half_period = 41;
59     set_alarm(half_period);
60
61     while(true) {
62         k_sem_take(&my_sem, K_FOREVER);
63         uint8_t *msg = ssm_rb_reader_claim(ssm_input_buffer);
64         if(msg) {
65             gpio_pin_set_dt(&led, *msg);
66             ssm_rb_reader_free(ssm_input_buffer);
67             set_alarm(half_period);
68         }
69     }
70 }
```

Fig. A9: The frequency generator implemented using Zephyr. When the timer goes off, the associated event (the new state of the LED) is placed in the ring buffer (line 38-41), and the main thread is woken up by releasing a semaphore (line 43). The main thread retrieves the event (line 63-66), and schedules the next wake-up point (line 67).

```haskell
{-# LANGUAGE ImplicitParams #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RebindableSyntax #-}
{-# OPTIONS_GHC -fplugin=SSM.Plugin -fplugin-opt=SSM.Plugin:mode=routine #-}
module Generator where

import Prelude
import SSM.Language
import SSM.Frontend.Peripheral.GPIO

-- signal generator, generating a signal on ?out0 with a half period specified by hpr
sig_gen :: (?out0 :: Ref GPIO) => Ref Time -> SSM ()
sig_gen hpr = routine $ while true $ do
    after (deref hpr) ?out0 (not' $ deref ?out0)
    wait ?out0

-- increments or decrements the half period by 2 us, depending on which of two buttons was pressed
button_handler :: (?b1 :: Ref Switch, ?b2 :: Ref Switch) => Ref Time -> SSM ()
button_handler hpr = routine $ while true $ do
    wait (?b1, ?b2)
    if changed ?b1 -- here we woke up, so either b1 or b2 was pressed
        then hpr <~ deref hpr + usecs 2
        else hpr <~ deref hpr - usecs 2

-- create shared half period reference and initialize it to 1 second, before spawning the concurrent child processes
entry :: (?out0 :: Ref GPIO, ?b1 :: Ref Switch, ?b2 :: Ref Switch) => SSM ()
entry = routine $ do
    hp <- var $ usecs 1000000
    fork [ sig_gen hp, button_handler hp]

-- Specify what inputs and outputs the program uses, and specify the initial contents on the ready queue
main :: Compile ()
main = do
    (led, handler) <- output 0
    button1 <- input 0
    button2 <- input 1

    let ?out0 = led
        ?b1 = button1
        ?b2 = button2

    schedule handler
    schedule entry
```

Fig. A10: A frequency generator implemented in Scoria. The half period of the generated frequency is adjusted in steps of 2 µs when button 0 or button 1 is pressed.

```c
#include "ssm-platform.h"
#ifndef SSM_DEBUG_TRACE
#define SSM_DEBUG_TRACE(...) \
    do                       \
        ;                    \
    while (0)
#endif
#ifndef SSM_DEBUG_MICROTICK
#define SSM_DEBUG_MICROTICK(...) \
    do                           \
        ;                        \
    while (0)
#endif
ssm_bool_t input0;
ssm_bool_t output1;
typedef char event;
typedef struct {
    struct ssm_act act;
    struct ssm_trigger trig1;
} act_b2b_t;
struct ssm_act *enter_b2b(struct ssm_act *caller, ssm_priority_t priority, ssm_depth_t depth);
void step_b2b(struct ssm_act *actg);
struct ssm_act *enter_b2b(struct ssm_act *caller, ssm_priority_t priority, ssm_depth_t depth) {
    struct ssm_act *actg = ssm_enter(sizeof(act_b2b_t), step_b2b, caller, priority, depth);
    act_b2b_t *acts = container_of(actg, act_b2b_t, act);

    acts->trig1.act = actg;
    return actg;
}
void step_b2b(struct ssm_act *actg) {
    act_b2b_t *acts = container_of(actg, act_b2b_t, act);

    SSM_DEBUG_TRACE("ActStepBegin␣\"b2b\"");
    SSM_DEBUG_MICROTICK();
    switch (actg->pc) {

    case 0:;
        while (true) {
            SSM_DEBUG_MICROTICK();
            SSM_DEBUG_TRACE("ActSensitize␣\"input0\"");
            ssm_sensitize(&(&input0)->sv, &acts->trig1);
            actg->pc = 1;
            return;

        case 1:;
            ssm_desensitize(&acts->trig1);
            ssm_assign_bool(&output1, actg->priority, (&input0)->value);
        }

    default:
        break;
    }
    ssm_leave(actg, sizeof(act_b2b_t));
}
int ssm_program_initialize(void) {
    ssm_initialize_bool(&input0);
    ssm_assign_bool(&input0, 0, 0);
    ssm_initialize_bool(&output1);
    ssm_assign_bool(&output1, 0, 0);
    bind_static_input_device((ssm_sv_t *)&input0.sv, 0U);
    ssm_activate(enter_b2b(&ssm_top_parent, SSM_ROOT_PRIORITY + 0 * (1 << SSM_ROOT_DEPTH - 1), SSM_ROOT_DEPTH - 1));
    ssm_activate(bind_static_output_device(&ssm_top_parent, SSM_ROOT_PRIORITY + 1 * (1 << SSM_ROOT_DEPTH - 1),
                                           SSM_ROOT_DEPTH - 1, &output1.sv, 0U));
    return 0;
}
```

Fig. A11: The generated C code from the Scoria program in Figure 15. For more details of code such as this, see [3].