Where Do Events Come From?

Reactive and Energy-Efficient Programming From The Ground Up

Anonymous Author(s)

Abstract

In reactive and event-based systems, execution is guided by an external environment which generates inputs to the application and consumes outputs from it. Reactive languages provide dedicated syntax and semantics to deal with events and greatly simplify the programming experience in this domain. Nevertheless, the environment is typically prefabricated in a host language and the very central concept of events is implemented externally to the reactive language. In this work, we propose an interrupt handler primitive for a reactive language targeting embedded systems in order to take control of the whole event loop, from a sensor input source and back to an actuator output. We propose the new asynchronous primitive in the context of the synchronous language Céu and discuss how they synergize to avoid race conditions at compile time, support lexically-scoped drivers, and provide automatic standby for applications.

Keywords interrupt service routine, sleep mode, synchronous reactive programming

1 Introduction

Reactive applications interact continuously and in real time with the external world through sensors and actuators (e.g., buttons, displays, timers, etc.). These interactions are typically represented as input events flowing from the devices to the application and as output events flowing from the application to the devices. As illustrated in Figure 1, devices can be encapsulated as a single component, *the environment*, which controls the system execution in an event loop: the application sits idle waiting for an input; the environment awakes the application on the occurrence of an input; the application reacts to the input and generates back one or more outputs; the application becomes idle and the loop restarts.

The environment is typically implemented in a host language (e.g., C) and controls the main event loop, invoking entry points in the reactive language runtime on the occurrence of inputs and also receiving output calls, both through a documented API. As examples, Esterel [2] relies on C for passing events between the environment and the running program [6], while Elm [4] uses the concept of *ports*, which allows sending out values

REBLS'18, November 04, 2018, Boston, MA, USA
2018.

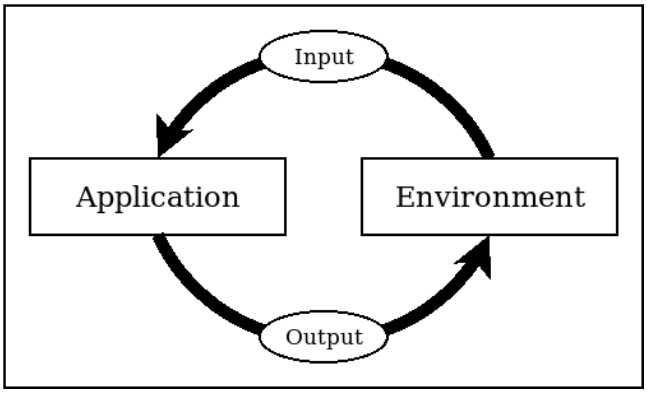

Figure 1. Event loop in reactive systems. The environment controls the application through input & output events.

to JavaScript in commands and listening for values in subscriptions [3].

The event-based interface between the application and the environment is arguably inevitable and also happens at the right level of abstraction, since it connects the application with the operating system resources through a non-invasive API. However, this separation exposes the environment as a rigid system component that evolves in separate from the application. It also requires two languages and the programmer has to deal with multiple syntaxes, incompatible type systems, and different address spaces. Furthermore, in the context of embedded systems, a proper host OS may even be absent or lacking enough device drivers, which requires more low-level intervention from the application.

In this work, we propose interrupt service routines (ISRs) as an asynchronous primitive for the synchronous language Céu [8]. ISRs empower applications to also implement their own device drivers and self-generate inputs, bypassing the need for a host environment. Céu targets resource-constrained architectures, such as Arduino-compatible microcontrollers, in which applications run in the bare metal, without operating system support. In this context, device drivers are typically libraries compiled together with the main program.

The lack of an OS opens an opportunity to explore a tighter integration between the application and its device drivers at the language level, resulting in an overall simplicity and tractability of the system. In particular, Céu is amenable to a simple static analysis to detect

race conditions that we extended to encompass ISRs. Céu also provides a lexical finalization mechanism that we adopt in drivers to properly disable interrupts. Finally, the synchronous semantics of Céu enforces that applications react to inputs in bounded time and remain in an idle states susceptible to standby. With the help of drivers and a power management runtime, Céu can put the microcontroller to sleep automatically at optimal sleeping modes after each input reaction.

We implemented an extensible runtime that exposes hooks for the interrupts and the power manager, which we validated in two microcontrollers: an *8-bit AVR/ATmega* and a *32-bit ARM/Cortex-M0*. We also implemented drivers for a variety of peripherals, such as GPIO, A/D converter, USART, SPI, and nRF24L01 transceiver. Applications can share data buffers with drivers to avoid unnecessary copying, with the static guarantee that no data races will occur. We demonstrate that applications built on top of these drivers show significant energy savings due to automatic standby.

TODO: weakness
what would be
output (int,int) O;
becomes
output (int x, int y) O do end
- excel define formulas for input but not the input itself
output -¿ input
the environment sits inverse reacts to output and generates inputs
remain idle and react to the occurrence of events

2 Céu: Structured Synchronous Reactive Programming

Céu is a Esterel-based [9] reactive programming language targeting resource-constrained embedded systems [8], with the characteristics that follow:

- **Reactive:** code only executes in reactions to events and is idle most of the time.
- **Structured:** programs use lexically-scoped structured control mechanisms, such as `await` (to suspend a line of execution), and `par` (to combine multiple lines of execution).
- **Synchronous:** reactions run atomically and to completion on each line of execution, i.e., there's no implicit preemption or real parallelism.

Structured reactive programming lets developers write code in direct style, recovering from the inversion of control imposed by event-driven execution [1, 5, 7].

The example in Listing 1 illustrates the main characteristics of Céu, namely event-driven I/O, lexically-scoped compositions and synchronous execution. The program toggles the state of the LED whenever a radio packet is received, terminating on a button press, always with the LED off. The program first declares the `LED` output, and the `BUTTON` and `RADIO_RECV` input events (ln 1–3). The declarations include a payload type, i.e., each event occurrence carries a value of that type (`high/low` is a boolean type). Then, the program uses a `par/or` composition to run two lines of execution, aka *trails*, in parallel: a single-statement trail that waits for a button press before terminating (ln 5), and an endless loop that toggles the LED on and off whenever a radio packet is received (ln 10–15). Since we care about any reception occurrence, the example ignores the actual packet payload (ln 12,14). The `finalize` clause (ln 7–9) ensures that, no matter how its enclosing trail terminates, the LED will be unconditionally turned off (ln 8).

```
1  output high/low LED;
2  input  high/low BUTTON;
3  input  Packet   RADIO_RECV;
4  par/or do
5      var high/low v = await BUTTON (until v==low);
6  with
7      finalize with
8          emit LED(low);
9      end
10     loop do
11         emit LED(high);
12         await RADIO_RECV;
13         emit LED(low);
14         await RADIO_RECV;
15     end
16 end
```

Listing 1. A Céu program that toggles the state of the LED whenever a radio packet is received, terminating on a button press, always with the LED off.

All communication between the application and the environment is done through the `await` and `emit` primitives, which awaits an input event and generates an output event, respectively.

The `par/or`, which stands for *parallel-or*, is a lexical composition that terminates as soon as one of the trails terminates, but which also automatically finalizes the other trail(s). In the example, when the button is pressed, not only the toggling loop will be aborted, but the `finalize` clause will turn the LED off unconditionally, since its enclosing block went out of scope.

The synchronous execution model of Céu dictates that reactions to input events are atomic and that incoming events are never lost. The event loop in Figure 1 indicates this behavior since the environment can only generate a new input after the application yields control and closes the loop. In the example, even if two

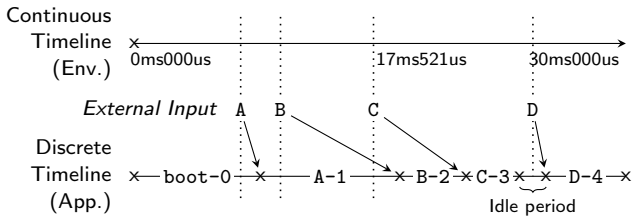

Figure 2. The discrete notion of time in Céu.

packets arrive simultaneously in the environment, the synchronous model provides the atomicity and responsiveness guarantees: the first `await` will awake and turn the LED off atomically (ln 12–13); and the second `await` will awake in sequence (ln 14).

Figure 2 illustrates the synchronous execution model of Céu. The continuous timeline in the environment shows an absolute reference clock with "physical timestamps" for the event occurrences (e.g., event `C` occurs at 17ms521us). The discrete timeline in the application shows how the same occurring events fit in the logical notion of time of Céu. The boot reaction `boot-0` happens before any input, at program startup. Event `A` "physically" occurs during `boot-0` but, because time is discrete, its corresponding reaction only executes afterwards, at logical instant `A-1`. Similarly, event `B` occurs during `A-1` and its reaction is postponed to execute at `B-2`. Event `C` also occurs during `A-1` but its reaction must also wait for `B-2` to execute and so it is postponed to execute at `C-3`. Finally, event `D` occurs during an idle period and can start immediately at `D-4`.

In order to guarantee responsiveness, the synchronous model relies on the *synchronous hypothesis*, which states that reactions must be significantly faster than the rate of inputs. For this reason, Céu (like most synchronous languages) restricts itself and refuses unbounded loops at compile time. This guarantees that all reactions to the environment are computed in bounded time [10], ensuring that applications are always responsive to incoming events.

2.1 Interrupt Service Routines

Interrupts service routines (ISRs) are software entry points that execute in response to hardware interrupts from peripherals such as timers and GPIOs. ISRs are at the lowest level interface between hardware and software and are the absolute source of inputs to programs. Typically, an ISR starts to execute as soon as the hardware interrupt occurs and suspends the normal program flow abruptly. Such asynchronous behavior reflects the inherent concurrent nature of peripherals interacting with the external world.

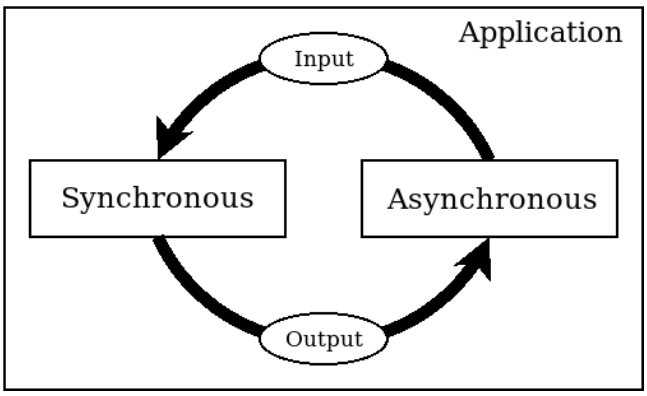

Figure 3. An application in Céu has a concurrent asynchronous side and a predictable synchronous side that receives and reacts to inputs atomically.

```
1 output high/low PIN_13;      // connected to an LED
2 input  high/low PIN_02;      // connected to a button
3 #include "gpio.ceu"
4
5 emit PIN_13(low);
6 loop do
7     var high/low v = await PIN_02;
8     emit PIN_13(v);
9 end
```

Listing 2. Synchronous application that turns the LED on whenever the button is pressed and off whenever it is unpressed.

We propose to extend Céu with asynchronous ISRs. However, asynchronous execution confronts the synchronous mindset of Céu since the assumption that reactions are atomic no longer holds. Not only this may lead to race conditions at a fine grain, but may also affect the ordering of events at a coarse grain: the effect of an earlier event may be perceived after the effect of a later event. Nevertheless, our goal is to preserve the well-behaved interaction between the Céu application and the environment as of Figure 1, even in the presence of asynchronous ISRs as illustrated in Figure 3: the original application is now represented by the synchronous Céu and the environment by the asynchronous Céu, all inside the same application. This way, we want to push all subtleties of asynchronous execution into device drivers which emit input events to regular synchronous code in Céu exactly as before.

Listing 2 and 3 implements an application that uses GPIO to link a button (pin 02) to an LED (pin 13) such that the LED is on whenever the button is pressed and off whenever it is unpressed.

The synchronous side (Listing 2) declares its interface with the external world and includes the asynchronous

```
1  // OUTPUT DRIVER
2
3  { pinMode(13, OUTPUT); }
4  output (high/low v) PIN_13 do
5      { digitalWrite(13, @v); }
6  end
7
8  // INPUT DRIVER
9
10 input high/low PIN_02;
11 {
12     pinMode(2, INPUT_PULLUP);
13     EICRA |= (1 << ISC00);   // sets INT0
14     EIMSK |= (1 << INT0);    // turns on INT0
15 }
16 spawn async/isr [INT0_vect] do
17     emit PIN_02({digitalRead(2)});
18 end
```

Listing 3. Asynchronous driver for GPIO which, on hardware interrupts, emits an input event into a queue.

side as a driver (ln 1–3). It then starts with the LED off (ln 5) and enters in a loop (ln 6–9) that, whenever the button changes (ln 7), changes the state of the LED with the new value (ln 8).

The asynchronous side (Listing 3) implements the output and input events. An `output` implementation (ln 4–6) is similar to a parameterized subroutine: whenever the application invokes `emit`, the output body executes atomically. Céu supports inline C between curly braces (ln 3,5) with interpolation to evaluates Céu expressions (e.g., `@v`). This allows drivers to take advantage of existing libraries in embedded toolchains, such as Arduino. In the example, the driver sets pin 13 as output (ln 3) when it is included, and sets its state whenever it is emitted (ln 5). An `input` event implementation requires an ISR registered with the `spawn async/isr` primitive (ln 16–18), which is automatically invoked whenever the associated interrupt occurs (e.g., `INT0_vect`). An ISR in Céu will typically have an `emit` to an input event inside its body to awake the synchronous side. However, although the ISR executes asynchronously as soon as the interrupt occurs, the `emit` respects the semantics of Céu and does not interrupt an ongoing reaction on the synchronous side (going into a queue to be discussed). In the example, the input driver first configures pin 2 to behave as input and to generate an external interrupt on level transitions (ln 11–15).

The example illustrates the clear separation between the application and the driver through an event-driven interface that reflects the desired architecture of Figures 1 and 3. Now the whole application is written in Céu: the synchronous side remains well behaved with no low-level calls, and the asynchronous side deals with all complexity in device drivers. Nevertheless, drivers are typically write-once components developed by embedded specialists, which are reused in most applications. Note that each architecture requires a mapping between the `async/isr` and the actual interrupt vector table, which we provide with our system.

As we discuss in the next section, supporting ISRs in the same language and memory space of the application allows programmers to choose between efficiency and safety during development by providing (and experimenting with) multiple interfaces with different levels of abstraction.

```
input void A;
input void B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```

Listing 4. Shared `x`

```
1  input void A;
2  // (empty line)
3  var int y = 1;
4  par/and do
5      await A;
6      y = y + 1;
7  with
8      await A;
9      y = y * 2;
10 end
```

Listing 5. Shared `y`

Figure 4. Accesses to shared `x` never concurrent. Accesses to shared `y` are concurrent but still deterministic.

,

2.2 Safe Shared-Memory Concurrency

In Céu, when multiple trails awake in the same reaction, they execute in lexical order, i.e., in the order they appear in the program source code. In Figure 4, both examples define a shared variable (ln 3) and assign to it in parallel trails (ln 6, 9). In Listing 4, the two assignments to `x` can only execute in reactions to different events `A` and `B` (ln 5,8), which cannot occur simultaneously in accordance with the synchronous model. Hence, for the sequence of events `A->B`, `x` becomes `4` (`(1+1)*2`), while for `B->A`, `x` becomes `3` (`(1*2)+1`). In Listing 5, the two assignments to `y` are simultaneous because they execute in reaction to the same event `A`. Nevertheless, because Céu follows lexical order, the execution is still deterministic, and `y` always becomes `4` (`(1+1)*2`).

Regardless of deterministic execution, note that an apparently innocuous change in the order of trails may modify the behavior of programs. To mitigate this threat, Céu performs concurrency checks at compile time to detect conflicting accesses to shared variables: if a variable is written in a trail segment, then a concurrent trail segment cannot access that variable [8]. The static

```
1  // USART INTERFACE
2
3  input none USART_RX;
4  var[32*] byte rx_buf;              // '*' = circular
5
6  // DRIVER
7
8  spawn async/isr [USART_RX_vect] do
9      rx_buf = rx_buf .. [{UDR0}];
10     emit USART_RX;
11 end
12
13 // APPLICATION
14
15 loop do
16     await USART_RX;
17     var int i;
18     loop i in [0 -> $rx_buf[ do    // '$' = length ($)
19         // uses rx_buf[i]
20     end
21     $rx_buf = 0;
22 end
```

Listing 6. The USART buffer is shared between the ISR and the application, resulting in a pontential race condition. *(The driver and application code are in separate files.)*

checks are optional (or raise warnings) and do not affect the semantics of the language.

The addition of asynchronous ISRs poses real threats concerning race conditions in programs since interrupts occur at arbitrary points. Listing 6 illustrates a minimum USART application (ln 13–22) that consumes incoming bytes (ln 18–21) as they arrive (ln 16). The USART API (ln 1–4) exposes an input event to signal incoming data (ln 3) and a circular buffer to avoid missing data (ln 4). The buffer is indispensable because the microcontroller cannot cope with the USART speed. The driver ISR (ln 6–11) simply appends incoming bytes to the end of the buffer (ln 9) and signal the application (ln 10). As a possible race condition, note that the ISR may fire and append a new byte to the buffer (ln 9) just before the application clears it (ln 21), in which case the new byte will be lost forever.

Céu treats an `async/isr` as a block of code that runs in parallel with the rest of the program (hence the required prefix `spawn`). This way, it is clear for the compiler that the accesses to `rx_buf` in Listing 6 may lead to a race condition (ln 9,21), and Céu raises a compile-time error. The programmer is responsible to protect concurrent memory accesses with `atomic` blocks, which disables interrupts (supposedly) for a short period of time.

```
1  // USART INTERFACE
2
3  code Usart_RX (var&[] byte buf, var int n) -> none;
4
5  // DRIVER
6
7  input none USART_RX;
8  var[32*] byte rx_buf;
9
10 spawn async/isr [USART_RX_vect] do
11     rx_buf = rx_buf .. [{UDR0}];
12     emit USART_RX;
13 end
14
15 code Usart_RX (var&[] byte buf, var int n) -> none
16 do
17     loop do
18         atomic do
19             buf = buf..rx_buf;
20             $rx_buf = 0;
21         end
22         if $buf >= n then
23             break;
24         end
25         await USART_RX;
26     end
27 end
28
29 // APPLICATION
30
31 loop do
32     var[255] byte buf;
33     await Usart_RX(&buf, 10);
34     // uses buf
35 end
```

Listing 7. The USART driver now exposes a safer (and more friendly) interface to the application. *(The driver and application code are in separate files.)*

Even so, we do not expect that application programmers are required to resolve race conditions. Céu supports reactive abstractions (similar to coroutines) that help hiding the concurrency complexity inside the driver. Listing 7 changes the USART API (ln 1–3) to expose a code abstraction that expects a reference to a buffer and a number of bytes to receive. Now, the application (ln 29–35) invokes the abstraction (ln 33) with a local buffer (ln 32). The driver (ln 5–27) now hides the low level interface of the previous example (ln 7–8) and implements the code abstraction (ln 15–27) that protects the access to the shared buffer with an `atomic` block (ln 18–21). The abstraction copies the driver buffer into the application buffer (ln 19) until the receiving the request number of bytes (ln 22). On the one hand, the

```
1  evt_t queue[MAX];            // input queue
2  void main () {
3    ceu_start();               // "boot reaction"
4    while (1) {
5      evt_t evt;
6      if (ceu_input(&evt)) {   // query input queue
7        ceu_sync(&evt);        // execute synchronous
8      } else {
9        ceu_pm_sleep();        // nothing to execute
10     }
11   }
12 }
```

Listing 8. XXX.

extra copying incurs an overhead, on the other hand, it prevents race conditions against the ISR.

2.3 Energy-Efficient Runtime

Our runtime of Céu implements the event loop proposed in Figure 3, which alternates between synchronous, well-behaved execution, and asynchronous ISRs. Since the language now has full control over the event loop and only executes from interrupts, we want the runtime to enter in sleep mode automatically to save energy.

Listing 8 is a realization of this architecture: The runtime defines an input queue (ln 1) in which ISRs enqueue new events (e.g., Listing 7, ln 12). In the `main` function, we first generate the *boot reaction* (ln 3), which starts the program to spawn the ISRs and reach the first `await` statements in the multiple lines of code. Then, the runtime enters an infinite loop (ln 4–11) that queries the input queue (ln 6) to awake the program (ln 7). Note that ISRs execute asynchronously with the loop, but they typically only emit an input event into the queue which will be queried in a subsequent iteration. Note also that if the queue is empty, the runtime enters in sleep mode to save energy (ln 9), and will only awake on a new hardware interrupt.

Microcontrollers typically support multiple levels of sleep mode, each progressively saves more energy at the expense of keeping less functionality active. As an example, the least efficient mode of the *ATmega328P* allows for timer interrupts since it keeps its internal clock active, while the most efficient mode turns off all peripherals and can only awake from external interrupts.

Our runtime supports three compile-time configurations for `ceu_pm_sleep` (ln 9): The first configuration is a *nop* which simply keeps the event loop running all the time without ever sleeping. This configuration is useful when introducing new platforms. The second configuration chooses a (inefficient) sleep mode that keeps all peripherals active. Although this configuration is not the most energy efficient, at least, it requires no extra assistance from the drivers. The third configuration keeps a bit vector of the active drivers during runtime and chooses the most efficient mode, querying the vector every time `ceu_pm_sleep` is called.

The third configuration achieves optimal efficiency but requires a tight interaction between the drivers and the power management runtime. Listing 9 unveils the power manager (ln 1–18), the modified USART driver (ln 20–29), and an illustrative application (ln 31–43). The

```
1  // POWER MANAGER (in C)
2
3  enum {
4      CEU_PM_ADC,
5      CEU_PM_TIMER1,
6      CEU_PM_USART,
7      ...
8  };
9
10 void ceu_pm_sleep (void) {
11     if (ceu_pm_get(CEU_PM_USART) || ...) {
12         sleep_mode_1(...);
13     } else if (ceu_pm_get(CEU_PM_ADC)) {
14         sleep_mode_2(...);
15     } else {
16         sleep_mode_3(...);
17     }
18 }
19
20 // USART DRIVER (in Ceu)
21
22 code Usart_RX (var&[] byte buf, var int n) -> none
23 do
24     {ceu_pm_set(CEU_PM_USART, 1);}
25     do finalize with
26         {ceu_pm_set(CEU_PM_USART, 0);}
27     end
28     ...    // same as in Listing 7
29 end
30
31 // APPLICATION (in Ceu)
32
33 input high/low PIN_02;       // connected to a button
34 loop do
35     par/or do
36         await PIN_02;
37     with
38         var[255] byte buf;
39         await Usart_RX(&buf, 10);
40         // uses buf
41     end
42     await PIN_02;
43 end
```

Listing 9. XXX. *(The power manager, driver and application code are in separate files.)*

application is a loop that awaits in parallel for a button press (ln 36) or receiving 10 bytes (ln 39). Let's call this initial state `STATE-A`. The `par/or` terminates when either of them occurs, going to the next line that awaits another button click (ln 42). Let's call this other state `STATE-B`. After the button click, the program loops back to `STATE-A`. While in `STATE-A`, the program can awake from USART and external interrupts, which means that the power manager should choose a sleep mode in which the USART remains operational. While in `STATE-B`, only external interrupts should awake the program, which menas that the power manager may use the most efficient sleep mode. The power manager enumerates all microcontroller's peripherals (ln 3–8) and, before sleeping, queries their states (ln 11,13) to choose the most appropriate sleep mode (ln 12,14,16). The USART driver needs to be extended with calls to enable and disable the USART state inside the power manager: just before awaiting, the driver enables the USART (ln 24) and registers a finalization block to unregister it on termination (ln 25–27). Termination may occur either directly after receiving the requested number of bytes, or indirectly if the `par/or` in the application terminates on a button click (ln 36).

Note that applications need not to be explicit about energy management to take advantage of sleep modes. All happens automatically because of the synchronous-reactive execution model and energy-aware runtime of Céu.

3 Related Work

- TinyOS - async, race condition avoidance - nesc manual - ICEM, standby - icem paper

- CRP

- finalize - structured programming - lexical scoping vs manual - no turn off

- citar ICEM o mais cedo nas duas ocasioes (async, energy) - citar TOS/ICEM na introducao como maior influencia - ja no abstract?

- falar que async/isr nao contem nada de Ceu, apenas shared memory e nocao de estar em paralelo - compartilhamento de memoria - velocidade - ajuda do compilador p/ detectar acesso concorrente

- finite is can be still much greater than 0 (order of milliseconds) - which is a lot of time

References

[1] A. Adya et al. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of ATEC'02*. USENIX Association, 289–302.

[2] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.

[3] Evan Czaplicki. 2018. Elm and JavaScript Interop. https://guide.elm-lang.org/interop/javascript.html (accessed in Jul-2018).

[4] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs.. In *Proceedings of PLDI'13*. 411–422.

[5] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. *Deprecating the observer pattern*. Technical Report.

[6] Dumitru Potop-Butucaru, Stephen A Edwards, and Gerard Berry. 2007. *Compiling esterel*. Vol. 86. Springer Science & Business Media.

[7] Guido Salvaneschi et al. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of Modularity'13*. ACM, 25–36.

[8] Francisco Sant'Anna et al. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

[9] Francisco Sant'anna et al. 2017. The Design and Implementation of the Synchronous Language Céu. *ACM TECS* 16, 4, Article 98 (July 2017), 26 pages. https://doi.org/10.1145/3035544

[10] Rodrigo C. M. Santos et al. 2018. A Memory-Bounded, Deterministic and Terminating Semantics for the Synchronous Programming Language CéU. In *Proceedings of LCTES'18*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3211332.3211334

A Appendix

Text of appendix ...