Where Do Events Come From?

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

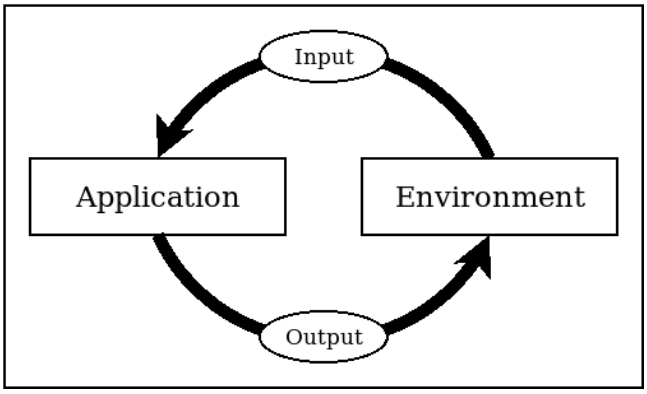

Figure 1. Event loop in reactive systems. The environment controls the application through input & output events.

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

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

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

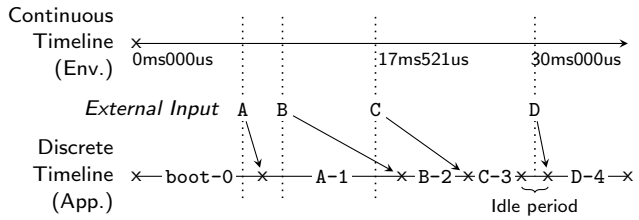

Figure 2. The discrete notion of time in Céu.

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.

3 Interrupt Service Routines in Céu

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.

Asynchronous execution confronts the synchronous mindset of Céu since the assumption that reactions

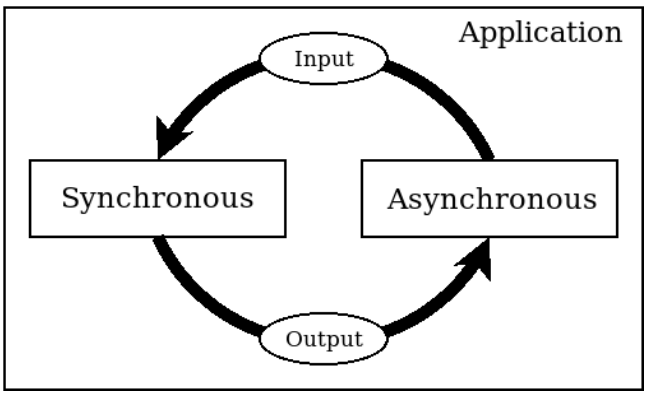

Figure 3. An application in Céu has a concurrent asynchronous side and a predictable synchronous side that receives and reacts to inputs atomically.

```
1 output high/low PIN_13;
2 input  high/low PIN_02;
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. XXX.

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. The synchronous side (Listing 2) declares its interface with the external world and includes the asynchronous side as a driver (ln 1–3). 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

```
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 = (EICRA & ~((1<<ISC00) | (1<<ISC01)))
14           | (CHANGE << ISC00);
15     EIMSK |= (1<<INT0);
16 }
17 spawn async/isr [INT0_vect] do
18     emit PIN_02({digitalRead(2)});
19 end
```

Listing 3. XXX.

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 17–19), which is automatically invoked whenever the associated interrupt occurs (e.g., `INT0_vect`). An ISR in Céu will always 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. In the example, the input driver first configures pin 2 to behave as input and to generate an external interrupt on level transitions (ln 11–16).

synchronous remains simple and well behaved with no low-level calls

The application only communicates

- 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 ...