Abstract

In reactive and event-based systems, execution is guided by an external environment that generates inputs to the application and is affected by 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 that targets embedded systems in order to take control of the whole event loop, from input generation up to output effects. We propose the new asynchronous primitive in the context of the synchronous language Céu and discuss how they synergize to prevent runtime race conditions at compile time, support lexically-scoped drivers, and provide automatic standby for applications.

Keywords interrupt service routines, reactive programming, standby, synchronous/asynchronous execution

1 Introduction

Reactive applications interact continuously and in real time with the external world through hardware peripherals such as sensors and actuators (e.g., buttons, displays, timers, etc.). These interactions are typically represented in software as input events flowing from the peripherals to the application and as output events flowing from the application to the peripherals. Peripherals can be abstracted in a single component, the environment, which connects with the application through an event loop: the application sits idle until the environment awakes it on an input; the application reacts and generates outputs back, which affect the environment; the application becomes idle and the loop restarts.

Reactive languages provide syntax and semantics specialized to deal with events and simplify the development of applications in this domain. However, the environment is still typically implemented in a host language (e.g., C) and controls the main event loop, invoking entry points into the reactive language runtime on the occurrence of inputs, and also receiving output calls from it.

The event-based interface between the reactive language and the environment is arguably inevitable, but reveals the environment as a rigid system component that evolves in separate from the application. It also requires the programmer to deal with multiple syntaxes, incompatible type systems, and different address spaces. Furthermore, in the context of embedded systems, a proper host operating system (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 reactive language Céu [10]. ISRs empower reactive applications with their own device drivers that self-generate inputs, bypassing the need for a host environment. Céu targets OS-less embedded architectures, such as Arduino-compatible microcontrollers. Applications in Céu can share data buffers with drivers, to avoid unnecessary copying, with some static guarantee that no data races will occur. Céu also provides a lexical finalization mechanism that safely disables drivers and turns off peripherals to save energy. Finally, the synchronous semantics of Céu enforces that applications react to inputs in bounded time and remain in idle states susceptible to standby. With the help of drivers and a power manager, the microcontroller sleeps automatically at optimal sleeping modes after each input reaction. We implemented drivers for a variety of peripherals, such as GPIO, A/D converter, USART, SPI, and an RF transceiver.

Our work is largely inspired by TinyOS [6], a low-power OS for sensor networks, which also provides asynchronous events with data race detection [5] and automatic energy management [7]. We adapted these ideas to the structured reactive model of Céu, and refined them with stronger guarantees due to its support for lexically scoped lines of execution with automatic finalization.

We start with an introduction of the structured synchronous reactive programming model of Céu (Section 2). Then, we propose to extend it with asynchronous ISRs and discuss how to interoperate with synchronous code through an event-driven interface (Section 3.1). We
also propose a simple static analysis to prevent race conditions between synchronous and asynchronous code at compile time (Section 3.2). Finally, we propose a new language runtime that enters in sleep mode automatically to save energy (Section 3.3).

2 Céu: Structured Synchronous Reactive Programming

Céu [10] is an Esterel-based [4] reactive programming language targeting resource-constrained embedded systems, with the characteristics that follow:

- **Reactive**: code only executes in reaction to events and is idle most of the time.
- **Structured**: programs use lexically-scoped structured control mechanisms such as `spawn` and `await` (to create and suspend 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, reversing from the inversion of control imposed by event-driven execution [1, 8, 9].

```plaintext
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 blinks the LED whenever a radio packet is received, terminating on a button press, always with the LED off.

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 synchronous execution model of Céu dictates that reactions to input events are atomic and that incoming events are never lost, which we refer to as the atomicity and responsiveness properties, respectively. In Listing 1, even if two packets arrive simultaneously in the environment, the synchronous model ensures these properties by adopting an event queue: the first `await` awakes from the first packet and turns the LED off atomically (ln 12–13); only after that, the second `await` will awake from the second packet (ln 14). From the point of view of the program, the packets arrive in sequence and each is handled in separate.

3 Interrupt Service Routines in Céu

In Listing 1, the input & output events are external to the program and are generated by the host language representing the environment. We now propose to extend Céu with interrupt services routines (ISRs) to take control of the whole event loop using a safe and energy-efficient approach.

3.1 Scoped ISRs

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 interface layer between the hardware and software and are the absolute source of inputs to programs. An ISR starts to execute as soon as a hardware interrupt occurs, suspending the ongoing program flow.
Where Do Events Come From?

1. output high/low PIN_13; // connected to LED
2. input high/low PIN_02; // connected to button
3. #include "gpio.ceu" // GPIO driver
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 code that turns the LED on whenever the button is pressed and off whenever it is unpressed.

abruptly. Such asynchronous behavior reflects the inherent concurrent nature of peripherals interacting with the real world.

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 might lead to race conditions at a fine grain, but might also affect the ordering of events at a coarse grain: the effect of an earlier event might be perceived after the effect of a later event. Still, our goal is to preserve the well-behaved interaction between CÉu and the environment even in the presence of asynchronous ISRs. Our approach is to push all subtleties of asynchronous execution into device drivers, which impersonate the environment and are responsible to emit input events towards regular synchronous code. Synchronous code preserves the atomicity and responsiveness properties, communicating with the environment only through events, exactly as before.

As a first example, Listing 2 and 3 use GPIOs to connect a button to an LED via software such that the LED (pin 13) is on whenever the button (pin 02) is pressed, and off whenever it is unpressed. The idea is to use Listing 2 handle synchronous/atomic changes in the button to affect the LED, and Listing 3 to handle asynchronous changes that are enqueued towards the synchronous side.

Listing 2, the synchronous side, first declares its input and output events. An output implementation (ln 4–6) is similar to a parameterized subroutine:

```
1  // OUTPUT DRIVER
2
3  { pinMode(13, OUTPUT); }
4  output (high/low v) PIN_13 do
5    { digitalWrite(13, @(v as int)); } end
6
7  // INPUT DRIVER
8
9  input high/low PIN_02;
10
11  { pinMode(2, INPUT_PULLUP); }
12  output (high/low) PIN_02 do
13    var high/low v = await PIN_02;
14    emit PIN_13(v);
15  end

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

whenever the application invokes emit, the output body executes atomically.

CÉu is designed to interoperate seamlessly with C and supports a compatible type system with self-evident conversion rules and same internal representation (e.g., high/low is a boolean, u8 is an unsigned 8-bit value, etc.). CÉu also supports inline C between curly braces (ln 3,5) and interpolation to evaluate CÉu expressions (e.g., @v).

In the example, when the driver is included, it configures pin 13 as output (ln 3) and sets its new state whenever PIN_13 is emitted (ln 5). An input event implementation uses an ISR registered with the spawn async/isr primitive (ln 16–18), which is automatically invoked whenever the associated interrupt occurs (e.g., INTO_vect). An ISR in CÉu will typically perform simple operations and emit an input event to awake the synchronous side (ln 17). However, although the ISR executes asynchronously when the interrupt occurs, the emit goes into an input queue and does not affect the synchronous side immediately. The input driver also configures pin 2 to behave as input and to generate external interrupts on level transitions (ln 11–15). The queue is static and new emissions of an unhandled input override the same position in the queue. This semantics is comparable to interrupt flags in microcontrollers and prevents buffer overflows.

The example illustrates the clear separation between the application and the driver through an event-driven interface. Now, the whole application is written in CÉu:
REBELS’18, November 04, 2018, Boston, MA, USA

Francisco Sant’Anna and Alexandre Sztajnberg

Listing 4. The A/D driver (ln 1–14) is included in the application (ln 21) inside a lexically-scoped block (ln 20–24), which turns off all driver functionalities automatically on termination. (The driver and application are actually in separate files.)

the synchronous side remains well behaved with no low-level calls, and the asynchronous side deals with the complexity of device drivers. Nevertheless, drivers are typically write-once components developed by embedded systems specialists, which are reused in most applications. Note that each supported architecture requires a mapping between the async/isr and the actual interrupt vector table (which we provide for the AVR/ATmega and ARM/Cortex microcontrollers).

The next example in Listing 4 illustrates the use of a lexically-scoped driver for the A/D converter. The application is a typical periodic sensor sampling loop, but which designates an explicit do-end block (ln 20–24) to include the driver (ln 21) and use it (ln 22–23). The driver specifies a finalization code (ln 6–12) that executes unconditionally whenever its enclosing block goes out of scope (ln 24). The finalize completely disables all A/D functionality to save energy (ln 8–10). As discussed in the previous section, the finalize would also execute if aborted from a hypothetical par/or enclosing the driver. Note that the driver interface (ln 3–4) is visible only inside the block (ln 20–24) and thus cannot be inadvertently used outside it.

3.2 Safe Shared-Memory Concurrency

In CÉU, when multiple trails awake in the same reaction, they execute atomically in lexical order, i.e., in the order they appear in the source code. In Figure 1, both listings define a shared variable (ln 3) which is assigned in parallel trails (ln 6, 9). In Listing 5, the two assignments to x can only execute in reactions to different events A and B (ln 3, 4), which cannot occur simultaneously due to the atomicity property of the synchronous model. In Listing 6, the two assignments to y are simultaneous because they execute in reaction to the same event A. Nevertheless, because CÉU follows lexical order and executes atomically, the outcome is still deterministic, and y always becomes 4 ((1+1)*2). Even so, CÉU performs (optional) 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 [10].

The addition of asynchronous ISRs now poses real threats concerning race conditions since they interrupt execution at arbitrary points and could possibly share memory with synchronous code. Listing 7 illustrates a minimum serial communication (USART) application (ln 15–24) that consumes incoming bytes (ln 19–23) as they arrive (ln 18). The USART API (ln 1–4) exposes an input event to signal incoming data (ln 3) and a buffer to prevent data loss (ln 4). The buffer is indispensable because the microcontroller might not cope with the USART speed. The (unsafe) driver ISR (ln 6–13) simply appends incoming bytes to the end of the buffer (ln 9) and signals the application (ln 11). As a possible race condition, the ISR might fire and append a new byte to the buffer (ln 9) just before the application clears it (ln 23), in which case the new byte would be lost forever.

Figure 1. Accesses to shared x never concurrent. Accesses to shared y are concurrent but still deterministic.


doi:10.23919/REBELS.2018.8582389

<table>
<thead>
<tr>
<th>Listing 5. Shared x</th>
<th>Listing 6. Shared y</th>
</tr>
</thead>
<tbody>
<tr>
<td>input void A;</td>
<td>input void A;</td>
</tr>
<tr>
<td>input void B;</td>
<td>// (empty line)</td>
</tr>
<tr>
<td>var int x = 1;</td>
<td>var int y = 1;</td>
</tr>
<tr>
<td>par/and do</td>
<td>par/and do</td>
</tr>
<tr>
<td>await A;</td>
<td>await A;</td>
</tr>
<tr>
<td>y = y + 1;</td>
<td>y = y + 1;</td>
</tr>
<tr>
<td>with</td>
<td>with</td>
</tr>
<tr>
<td>await B;</td>
<td>await A;</td>
</tr>
<tr>
<td>x = x * 2;</td>
<td>y = y * 2;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Listing 5. Shared x Listing 6. Shared y

<table>
<thead>
<tr>
<th>Listing 6.</th>
<th>Shared y</th>
</tr>
</thead>
<tbody>
<tr>
<td>input void A;</td>
<td>input void A;</td>
</tr>
<tr>
<td>input void B;</td>
<td>// (empty line)</td>
</tr>
<tr>
<td>var int x = 1;</td>
<td>var int y = 1;</td>
</tr>
<tr>
<td>par/and do</td>
<td>par/and do</td>
</tr>
<tr>
<td>await A;</td>
<td>await A;</td>
</tr>
<tr>
<td>y = y + 1;</td>
<td>y = y + 1;</td>
</tr>
<tr>
<td>with</td>
<td>with</td>
</tr>
<tr>
<td>await B;</td>
<td>await A;</td>
</tr>
<tr>
<td>y = y * 2;</td>
<td>y = y * 2;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Listing 6. Shared y

<table>
<thead>
<tr>
<th>Listing 7</th>
<th>Minimum Serial Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>input void A;</td>
<td>input void A;</td>
</tr>
<tr>
<td>input void B;</td>
<td>// (empty line)</td>
</tr>
<tr>
<td>var int x = 1;</td>
<td>var int y = 1;</td>
</tr>
<tr>
<td>par/and do</td>
<td>par/and do</td>
</tr>
<tr>
<td>await A;</td>
<td>await A;</td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>y = y + 1;</td>
</tr>
<tr>
<td>with</td>
<td>with</td>
</tr>
<tr>
<td>await B;</td>
<td>await A;</td>
</tr>
<tr>
<td>x = x * 2;</td>
<td>y = y * 2;</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Listing 7. Minimum Serial Communication
Where Do Events Come From?

Listing 7. The USART buffer is shared between the ISR and the application, resulting in a potential race condition. (The driver and application are actually in separate files.)

CÉu treats a spawn async/isr as a block that runs in parallel with the rest of the program (hence the prefix spawn). This way, it is clear for the compiler that the accesses to rx_buf in Listing 7 may lead to race conditions (e.g., Ln 9,23), and CÉu raises a compile-time error. The programmer is responsible to protect concurrent memory accesses with atomic blocks, which disables interrupts for a short period of time.

However, we do not expect that application programmers should be required to resolve race conditions. CÉu provides code reactive abstractions (similar to coroutines) that help hiding concurrency issues inside drivers. Listing 8 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) passing a local buffer (ln 32). The driver (ln 5–27) now hides the low-level interface (ln 7–8) and implements the abstraction (ln 15–27) that protects the accesses to the shared buffer with an atomic block (ln 18–21). The code copies the driver buffer into the application buffer (ln 19) up to the requested number of bytes (ln 22). On the one hand the extra copying incurs a memory and runtime overhead, but on the other hand it prevents race conditions with the ISR.

Listing 8. The USART driver now exposes a safer (and more friendly) interface to the application. (The driver and application are actually in separate files.)

Support for ISRs in the same language and memory space of the application allows programmers to choose between efficiency and safety during development by providing multiple interfaces with different levels of abstraction.

3.3 Energy-Efficient Runtime

The language runtime now alternates between synchronous, well-behaved execution, and asynchronous, unpredictable ISRs. Since programs only execute from interrupts, the runtime can now also enter in sleep mode automatically to save energy, as depicted in Listing 9: It first defines an input queue (ln 1) in which ISRs enqueue new events (e.g., Listing 8, ln 12). The main function generates the boot reaction once to start the program (ln 3),
Listing 9. Overall runtime architecture of CÉu with an input queue (ln 1), event loop (ln 4–11), synchronous execution (ln 7), and standby mode (ln 9).

which will spawn the ISRs and reach the first await statements in the multiple trails of the program. Then, the runtime enters the event loop (ln 4–11), which queries the input queue (ln 6) to awake the program (ln 7). ISRs execute asynchronously, emitting input events into the queue, which will be queried in subsequent iterations of the loop. Note 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 modes, each progressively saves more energy at the expense of keeping less functionality active. As an example, the least efficient mode of the ATmega328P [2] 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 (e.g., GPIO falling edge).

Our runtime supports three compile-time configurations for ceu_pm_sleep (ln 9). The first configuration is a nop that 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 sleep configuration keeps a bit vector of the active drivers during runtime and chooses the most efficient mode, querying this 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 manager. Listing 10 unveils the power manager (ln 1–18), the modified USART driver (ln 20–29), and an illustrative application (ln 31–43). The application is a loop that initially awaits in parallel for a button press (ln 36) or receiving 10 bytes (ln 39). Let’s call this 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 another 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 means 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 now needs to be extended with calls to enable and disable the USART state inside the power manager: just before awaiting (ln 28), the driver enables the USART (ln 24) and creates a finalization block to disable 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 never need to be explicit about energy management to take advantage of sleep modes. All happens automatically because of the synchronous reactive execution model and the interaction between the drivers and energy-aware runtime of Céu. Our initial tests show optimal energy savings for applications in idle states [11].

4 Conclusion

We extend the synchronous language Céu with asynchronous interrupt service routines (ISRs) to take control of the whole event loop in reactive systems, from input generation up to output effects. ISRs empower reactive applications to also implement their own device drivers and self-generate inputs, bypassing the need for a host environment. However, asynchronous execution confronts the well-behaved semantics of synchronous languages with possible race conditions. To mitigate this threat, we extend the static concurrency checks of Céu to also detect data races between ISRs and the application.

Our approach for developing drivers relies on the lexical structured mechanisms of Céu, such as parallel compositions and abortion of lines of execution, to offer stronger guarantees. For instance, the visibility of drivers can be delimited to scoped blocks to avoid resource leaks. The same mechanism allows drivers to signal the power manager that associated peripherals are no longer required, allowing applications to use optimal sleep modes automatically.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. mmmmm and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References