VII
The implementation of Céu

The compilation process of a program in CÉU is composed of three main phases, as illustrated in Figure VII.1:

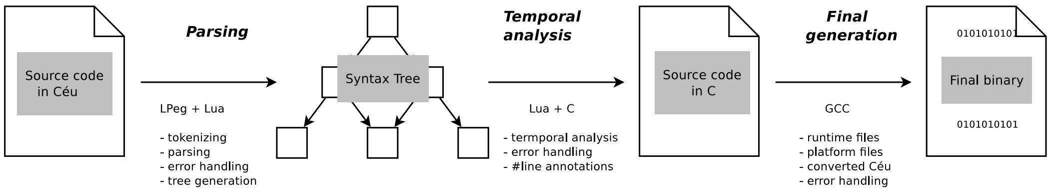

Figure VII.1: Compilation process: from the source code in CÉU to the final binary.

Parsing The parser of CÉU is written in *LPeg* [24], a pattern matching library that also recognize grammars, making it possible to write the tokenizer and grammar with the same tool. The source code is then converted to an *abstract syntax tree (AST)* to be used in further phases. This phase may be aborted due to syntax errors in the CÉU source file.

Temporal analysis This phase detects inconsistencies in CÉU programs, such as unbounded loops and the forms of non-determinism. It also makes some "classical" semantic analysis, such as building a symbol table for checking variable declarations. However, most of type checking is delayed to the last phase to take advantage of GCC's error handling. Therefore, this phase needs to annotate the *C* output with `#line` pragmas that match the original file in CÉU. This phase must output code in *C*, given how tied CÉU is to *C* by design.

Final generation The final phase packs the generated *C* file with the CÉU runtime and platform-dependent functionality, compiling them with *gcc* and generating the final binary. The CÉU runtime includes the scheduler, timer management, and the external *C* API. The platform files include libraries for I/O and bindings to invoke the CÉU scheduler on external events.

In the sections that follow, we discuss the most sensible parts of the compiler considering our design, such as the temporal analysis, runtime scheduler, and the external API.

VII.1 Temporal analysis

As introduced, the *temporal analysis* phase detects inconsistencies in Céu programs. Here, we focus on the algorithm that detects non-deterministic access to variables, as presented in Section III.2.

For each node representing a statement in the program AST, we keep the set of events I (for *incoming*) that can lead to the execution of the node, and also the set of events O (for *outgoing*) that can terminate the node.

A node inherits the set I from its direct parent and calculates O according to its type:

- Nodes that represent expressions, assignments, C calls, and declarations simply reproduce $O = I$, as they do not await;
- An `await e` statement has $O = \{e\}$.
- A `break` statement has $O = \{\}$ as it escapes the innermost `loop` and never terminate, i.e., never proceeds to the statement immediately following it (see also `loop` below);
- A *sequence node (;)* modifies each of its children to have $I_n = O_{n-1}$. The first child inherits I from the sequence parent, and the set O for the sequence node is copied from its last child, i.e., $O = O_n$.
- A `loop` node includes its body's O on its own I ($I = I \cup O_{body}$), as the loop is also reached from its own body. The union of all `break` statements' O forms the set O for a `loop`.
- An `if` node has $O = O_{true} \cup O_{false}$.
- A parallel composition (`par/and` / `par/or`) may terminate from any of its branches, hence $O = O_1 \cup ... \cup O_n$.

With all sets calculated, any two nodes that perform side effects and are in parallel branches can have their I sets compared for intersections. If the intersection is not the empty set, they are marked as suspicious (see Section III.2).

Figure VII.2 reproduces the second code of Figure III.5 and shows the corresponding *AST* with the sets I and O for each node. The event `.` (dot) represents the "boot" reaction. The assignments to `y` in parallel (lines 5,8 in the code) have an empty intersection of I (lines 6,9 in the AST), hence, they do not conflict. Note that although the accesses in lines 5, 11 in the code (lines 6,11 in the AST) do have an intersection, they are not in parallel and are also safe.

```
1  input void A, B;
2  var int y;
3  par/or do
4    await A;
5    y = 1;
6  with
7    await B;
8    y = 2;
9  end
10 await A;
11 y = 3;
```

```
1  Stmts I={.} O={A}
2      Dcl_y I={.} O={.}
3      ParOr I={.} O={A,B}
4          Stmts I={.} O={A}
5              Await_A I={.} O={A}
6              Set_y I={A} O={A}
7          Stmts I={.} O={B}
8              Await_B I={.} O={B}
9              Set_y I={B} O={B}
10     Await_A I={A,B} O={A}
11     Set_y I={A} O={A}
```

Figure VII.2: A program with a corresponding AST describing the sets I and O. The program is safe because accesses to `y` in parallel have no intersections for I.

VII.2 Memory layout

Céu favors a fine-grained use of trails, being common the use of trails that await a single event. For this reason, Céu does not allocate per-trail stacks; instead, all data resides in fixed memory slots—this is true for the program variables as well as for temporary values and flags needed during runtime. Memory for trails in parallel must coexist, while statements in sequence can reuse it. Céu reserves a single static block of memory to hold all memory slots, whose size is the maximum the program uses at a given time. A given position in the memory may hold different data (with variable sizes) during runtime.

Translating this idea to *C* is straightforward [28, 5]: memory for blocks in sequence are packed in a `struct`, while blocks in parallel, in a `union`. As an example, Figure VII.3 shows a program with corresponding memory layout. Each variable is assigned a unique *id* (e.g. `a_1`) so that variables with the same name can be distinguished. The `do-end` blocks in sequence are packed in a `union`, given that their variables cannot be in scope at the same time, e.g., `MEM.a_1` and `MEM.b_2` can safely share the same memory address. The example also illustrates the presence of runtime flags related to the parallel composition, which also reside in reusable slots in the static memory.

VII.3 Trail allocation

The compiler extracts the maximum number of trails a program can have at the same time and creates a static vector to hold runtime information about

```
input int A, B, C;
do
    var int a = await A;
end
do
    var int b = await B;
end
par/and do
    await B;
with
    await C;
end
```

```
union {                 // sequence
    int a_1;            //   do_1
    int b_2;            //   do_2
    struct {            //   par/and
        u8 _and_3: 1;
        u8 _and_4: 1;
    };
} MEM ;
```

Figure VII.3: A program with blocks in sequence and in parallel, with corresponding memory layout.

them. Again, trails that cannot be active at the same time can share memory slots in the static vector.

At any given moment, a trail can be awaiting in one of the following states: `INACTIVE`, `STACKED`, `FIN`, or in any event defined in the program:

```
enum {
    INACTIVE = 0,
    STACKED,
    FIN,
    EVT_A,       // input void A;
    EVT_e,       // event int e;
    <...>        // other events
}
```

All terminated or not-yet-started trails stay in the `INACTIVE` state and are ignored by the scheduler. A `STACKED` trail holds its associated stack level and is delayed until the scheduler runtime level reaches that value again. A `FIN` trail represents a hanged finalization block which is only scheduled when its corresponding block goes out of scope. A trail waiting for an event stays in the state of the corresponding event, also holding the sequence number (*seqno*) in which it started awaiting. A trail is represented by the following `struct`:

```
struct trail_t {
    state_t evt;
    label_t lbl;
    union {
        unsigned char seqno;
        stack_t       stk;
    };
};
```

The field `evt` holds the state of the trail (or the event it is awaiting); the field `lbl` holds the entry point in the code to execute when the trail is

```
1  input void A;
2  event void e;
3  // TRAIL 0 — lbl Main
4  par/and do
5    // TRAIL 0 — lbl Main
6    await e;
7    // TRAIL 0 — lbl Awake_e
8    // TRAIL 0 — lbl ParAnd_chk
9  with
10   // TRAIL 1 — lbl ParAnd_sub_2
11   await A;
12   // TRAIL 1 — lbl Awake_A_1
13   emit e;
14   // TRAIL 1 — lbl Emit_e_cont
15   // TRAIL 1 — lbl ParAnd_chk
16 end
17 // TRAIL 0 — lbl ParAnd_out
18 await A;
19 // TRAIL 0 — lbl Awake_A_2
```

```
enum {
  Main = 1,        // ln  3
  Awake_e,         // ln  7
  ParAnd_chk,      // ln  8, 15
  ParAnd_sub_2,    // ln 10
  Awake_A_1,       // ln 12
  Emit_e_cont,     // ln 14
  ParAnd_out,      // ln 17
  Awake_A_2        // ln 19
};
```

Figure VII.4: Static allocation of trails and entry-point labels.

scheduled; the third field depends on the `evt` field and may hold the `seqno` for an event, or the stack level `stk` for a `STACKED` state.

The size of `state_t` depends on the number of events in the application; for an application with less than 253 events (plus the 3 states), one byte is enough. The size of `label_t` depends primarily on the number of `await` statements in the application—each `await` splits the code in two and requires a unique entry point in the code for its continuation. Additionally, split & join points for parallel compositions, `emit` continuations, and finalization blocks also require labels. The `seqno` will eventually overflow during execution (every 256 reactions). However, given that the scheduler traverses all trails in each reaction, it can adjust them to properly handle overflows (actually 2 bits to hold the `seqno` would be already enough). The stack size depends on the maximum depth of nested emissions and is bounded to the maximum number of trails, e.g., a trail emits an event that awakes another trail, which emits an event that awakes another trail, and so on—the last trail cannot awake any trail, because they will be all hanged in a `STACKED` state. In WSNs applications, the size of `trail_t` is typically only 3 bytes (1 byte for each field).

(a) Code generation

The example in Figure VII.4 illustrates how trails and labels are statically allocated in a program. The program has a maximum of 2 trails, because the `par/and` (line 4) can reuse *TRAIL 0*, and the join point (line 16) can reuse both *TRAIL 0* and *TRAIL 1*. Each label is associated with a unique identifier

```
1  while (<...>) {                 // scheduler main loop
2     trail_t* trail = <...>   // choose next trail
3     switch (trail->lbl) {
4        case Main:
5           // activate TRAIL 1 to run next
6           TRLS[1].evt = STACKED;
7           TRLS[1].lbl = ParAnd_sub_2;  // 2nd trail of par/and
8           TRLS[1].stk = current_stack;
9
10          // code in the 1st trail of par/and
11          // await e;
12          TRLS[0].evt = EVT_e;
13          TRLS[0].lbl = Awake_e;
14          TRLS[0].seq = current_seqno;
15          break;
16
17       case ParAnd_sub_2:
18          // await A;
19          TRLS[1].evt = EVT_A;
20          TRLS[1].lbl = Awake_A_1;
21          TRLS[1].seq = current_seqno;
22             break;
23
24         <...>   // other labels
25     }
26 }
```

Figure VII.5: Generated code for the program of Figure VII.4.

in the `enum`. The static vector to hold the two trails in the example is defined as

```
trail_t TRLS[2];
```

In the final generated *C* code, each label becomes a *switch case* working as the entry point to execute its associated code. Figure VII.5 shows the corresponding code for the program of Figure VII.4. The program is initialized with all trails set to `INACTIVE`. Then, the scheduler executes the *Main* label in the first trail. When the *Main* label reaches the `par/and`, it "stacks" the 2nd trail of the `par/and` to run on *TRAIL 1* (line 5-8) and proceeds to the code in the 1st trail (lines 10-15), respecting the deterministic execution order. The code sets the running *TRAIL 0* to await `EVT_e` on label `Awake_e`, and then halts with a `break`. The next iteration of the scheduler takes *TRAIL 1* and executes its registered label `ParAnd_sub_2` (lines 17-22), which sets *TRAIL 1* to await `EVT_A` and also halts.

Regarding cancellation, trails in parallel are always allocated in subsequent slots in the static vector `TRLS`. Therefore, when a `par/or` terminates, the scheduler sequentially searches and executes `FIN` trails within the range of the `par/or`, and then clears all of them to `INACTIVE` at once. Given that finalization blocks cannot contain `await` statements, the whole process is guaranteed to terminate in bounded time. Escaping a `loop` that contains parallel compositions also trigger the same process.

VII.4 The external *C* API

As a reactive language, the execution of a program in CÉU is guided entirely by the occurrence of external events. From the implementation perspective, there are three external sources of input into programs, which are all exposed as functions in a *C* API:

ceu_go_init(): initializes the program (e.g. trails) and executes the "boot" reaction (i.e., the `Main` label).

ceu_go_event(id,param): executes the reaction for the received event id and associated parameter.

ceu_go_wclock(us): increments the current time in microseconds and runs a reaction if any timer expires.

Given the semantics of CÉU, the functions are guaranteed to take a bounded time to execute. They also return a status code that says if the CÉU program has terminated after the reactions. Further calls to the API have no effect on terminated programs.

The bindings for the specific platforms are responsible for calling the functions in the API in the order that better suit their requirements. As an example, it is possible to set different priorities for events that occur concurrently (i.e. while a reaction chain is running). However, a binding must never interleave or run multiple functions in parallel. This would break the CÉU sequential/discrete semantics of time.

As an example, Figure VII.6 shows our binding for *TinyOS* which maps *nesC* callbacks to input events in CÉU. The file `ceu.h` (included in line 3) contains all definitions for the compiled CÉU program, which are further queried through `#ifdef`'s. The file `ceu.c` (included in line 4) contains the main loop of CÉU pointing to the labels defined in the program. The callback `Boot.booted` (lines 6-11) is called by TinyOS on mote startup, so we initialize CÉU inside it (line 7). If the CÉU program uses timers, we also start a periodic timer (lines 8-10) that triggers callback `Timer.fired` (lines 13-17) every 10 milliseconds and advances the wall-clock time of CÉU (line 15)[1]. The remaining

[1] We also offer a mechanism to start the underlying timer on demand to avoid the "battery unfriendly" 10ms polling.

lines map pre-defined TinyOS events that can be used in CÉU programs, such as the light sensor (lines 19-23) and the radio transceiver (lines 25-36).

```
implementation
{
    #include "ceu.h"
    #include "ceu.c"

    event void Boot.booted () {
        ceu_go_init();
#ifdef CEU_WCLOCKS
        call Timer.startPeriodic(10);
#endif
    }

#ifdef CEU_WCLOCKS
    event void Timer.fired () {
        ceu_go_wclock(10000);
    }
#endif

#ifdef _EVT_PHOTO_READDONE
    event void Photo.readDone (uint16_t val) {
        ceu_go_event(EVT_PHOTO_READDONE, (void*)val);
    }
#endif

#ifdef _EVT_RADIO_SENDDONE
    event void RadioSend.sendDone (message_t* msg) {
        ceu_go_event(EVT_RADIO_SENDDONE, msg);
    }
#endif

#ifdef _EVT_RADIO_RECEIVE
    event message_t* RadioReceive.receive (message_t* msg) {
        ceu_go_event(EVT_RADIO_RECEIVE, msg);
        return msg;
    }
#endif

    <...>   // other events
}
```

Figure VII.6: The *TinyOS* binding for Céu.