CÉU-MEDIA: A Multimedia Library for the Synchronous Language CÉU

ABSTRACT
We investigate the use of the synchronous language CÉU for programming multimedia applications, in particular, those applications that can be described as a set of synchronized media objects. The result of this investigation is CÉU-MEDIA, a library for programming multimedia CÉU. The programming model and abstractions of CÉU-MEDIA are similar to that of the traditional high-level multimedia languages NCL and SMIL, but avoids their inflexibility, ambiguity, and synchronization problems. This is possible because CÉU-MEDIA takes full advantage of CÉU features: its integration with C, its abstraction mechanisms, and its semantics. And because, its implementation ensures that the properties CÉU semantics are reflected in the output multimedia presentation. The paper compares the synchronization paradigm of CÉU with those of NCL and SMIL, discusses the implementation of CÉU-MEDIA, and validates the proposal by examining the implementation of some representative use cases.

CCS Concepts
• Software and its engineering → Development frameworks and environments; Application specific development environments;

Keywords
Multimedia; CÉU; CÉU-MEDIA; Inter-media synchronization; Synchronous Hypothesis

1. INTRODUCTION
We present CÉU-MEDIA, a library for authoring multimedia applications using the synchronous language CÉU. With CÉU-MEDIA authors describe multimedia presentations in an abstraction level close to that adopted by traditional high-level multimedia languages, such as NCL or SMIL, while avoiding most of their limitations and pitfalls (inflexibility, ambiguity, and logical and physical dyssynchrony). Why another multimedia library? Because current libraries (GStreamer, FFmpeg, QT-Multimedia, LibVLC, etc.) are too low-level; they assume specialist users and rely on complex programming models. And also because there are few proposals that try to apply the synchronous approach to the problem of multimedia synchronization—at least at the level of abstraction we are considering.

The first advantage of using CÉU-MEDIA is its flexibility. Under the hood one has a full-fledged programming language: using CÉU constructs, one can create abstractions more suitable to particular scenarios by combining those already defined by CÉU-MEDIA. Flexibility often comes with the price of added complexity, but that is a price one should be willing to pay when extending a language. In contrast, NCL and SMIL are inflexible languages: any extension must be done externally via pre-processors or via scripts (Lua or JavaScript) that modify the original program.

The second advantage of CÉU-MEDIA is the straightforward, accurate semantics induced by the synchronous hypothesis and enforced determinism. CÉU is a synchronous language with a precise semantics: a program reacts to external events in a way that these reactions are conceptually instantaneous and always deterministic. The passage of time is represented by an ordinary event and can be controlled precisely by programs. This precise treatment of (logical) time is essential to the description of any synchronization scenario, and especially to those occurring in multimedia. In contrast, the semantics of NCL and SMIL programs is notoriously ambiguous and inconsistent.

The third and last advantage of CÉU-MEDIA we discuss in the paper has to do with how its multimedia concepts are implemented. The library was built on top of GStreamer, which is an industry-grade framework for the construction of multimedia systems. In its implementation, we strove to maintain as much as possible the accuracy imposed by the synchronous semantics of CÉU. For instance, in CÉU, time passes only when the program says so, which means that, when programming with CÉU-MEDIA, audio and video samples are generated only when the program says so. This precise control of the output presentation cannot be achieved (not even specified) in NCL and SMIL, or similar languages.

The rest of the paper is organized as follows. In Section 2 we present briefly the CÉU programming language. In Section 3 we compare the general synchronization constructs of CÉU with those of NCL and SMIL. In Section 4 we present the architecture and implementation of CÉU-MEDIA. In Section 5 we discuss some use cases and examine their implementation in CÉU-MEDIA. Finally, in Section 6 we draw our conclusions and point out future work.
2. CÉU IN A NUTSHELL

CÉU [14] is a synchronous programming language for developing safe concurrent programs. By synchronous, we mean that its programs assume the synchronous hypothesis [3], i.e., that program reactions are conceptually instantaneous and always terminate. Added to this hypothesis, pure CÉU programs are by definition deterministic, hence the adjective "safe". If we view a CÉU program as a blackbox that reacts to external events, then the synchronous part guarantees that such reactions are instantaneous (from the point of view of program logic), while the deterministic part guarantees that the occurrence of an event in a given program state always leads to the same final state.

Determinism is a desirable property of systems in general, but it is even more desirable when concurrency is involved—nondeterministic, concurrent programs are a profuse source of bugs, they are often harder to compose, debug, and analyze than their deterministic counterparts [4]. In CÉU, concurrency can only be programmed via the compositions par, par/or and par/and, which create concurrent execution trails when evaluated. The execution of such trails is necessarily deterministic and the CÉU compiler enforces mutual exclusion between them so that access to shared variables is always consistent [14].

To illustrate these concepts, consider the CÉU program depicted in Listing 1. This program blinks two LEDs, Led1 and Led2, by changing their state (on or off) every couple of seconds. When the program starts, the LEDs go on blinking until a key is pressed, i.e., event KEY occurs, at which point the program terminates.

```
input void KEY;
par/or
    do /* trail 1 */
        loop do
            _Led1_off()
            await 2s;
            _Led1_on()
            await 2s;
        end
    with /* trail 2 */
        loop do
            _Led2_on()
            await 4s;
            _Led2_off()
            await 4s;
        end
    with /* trail 3 */
        await KEY;
end
```

Listing 1. Blinking LEDs in CÉU.

In Listing 1 line 1 declares the external input event KEY. Lines 2–19 define a parallel composition having 3 trails. The first trail (lines 4–9) executes an infinite loop that awaits 2 seconds, turns Led1 off, waits 2 more seconds and turns it on. The second trail (lines 11–16) is similar, but it awaits 4 seconds to turn Led2 on and off. The last trail (line 18) awaits for event KEY and terminates. When the program starts, the three trails are started; trails 1 and 2 run indefinitely, blinking their corresponding LEDs with the programmed frequency, while trail 3 simply waits for a KEY before terminating. Because the par/or composition ends when any of its trails end, the three trails will join at line 19 when trail 3 terminates with a key press.

Note that CÉU trails are not operating-system threads. OS threads can be preempted at any time by the scheduler, which often leads to nondeterminism and synchronization problems. In contrast, the CÉU compiler generates a single-threaded program that schedules the execution of its trails in a completely deterministic manner. The trail-scheduling algorithm of CÉU can be summarized in four steps:

1. The program initiates with a single trail.
2. Then its active trails execute until they block (wait for some external input event) or terminate.
3. When all trails block, which inevitably happens due to the synchronous hypothesis, the reaction is done; the program goes idle and the environment takes control.
4. If an external input event occurs, the environment gives control back to the program; all trails that are blocked waiting for event are resumed, and we are back in step (2).

Figure 1 depicts a timeline representing the state of the LEDs of the program in Listing 1. The synchronous and deterministic execution model of CÉU guarantees that the pattern presented in this 18-second timeline is repeated indefinitely until some key is pressed by the user. Every 4 seconds, the program executes three function calls in exactly the same order. First, it turns Led1 on and off (lines 6 and 8 of Listing 1), and then either turns Led2 on (line 13) or off (line 15). From the program’s perspective, these calls are simultaneous; they occur in the same reaction, i.e., both trails react to the same event (viz., the passage of 4 seconds), and therefore (logical) time does not pass between the calls.

![Figure 1. Timeline of the blinking LEDs program.](image)

Given an input program such as that of Listing 1, the CÉU compiler generates a corresponding C program. In this process, it checks for inconsistencies and makes sure that the properties advertised by the semantics of CÉU (synchronicity, termination, consistency and determinism) are reflected in the resulting C logic. The exceptions are native C calls, which are the statements starting with an underscore (_), e.g., lines 6, 8, 13 and 15 in Listing 1. These are mapped directly into C calls which cannot be checked by the compiler. The drawback here is that if a native call performs a blocking operation, i.e., one that takes a non-negligible time to return, the logical time may diverge from the physical time. For instance, the "2s" written in the CÉU program may not correspond exactly to two physical seconds (though they will always mean two logical seconds, i.e., two occurrences of event “second”). That said, for our purposes this is not a big problem. We expose a high-level pure CÉU API to application authors, namely, CÉU-MEDIA, so that in general they do not write custom C code.

3. COMPARING CÉU TO NCL AND SMIL

CÉU-MEDIA aims to describe multimedia presentations in a strictly precise way in both dimensions logical and physical, i.e., from the point of view of the program state and
the resulting audio and video samples. To validate the CÉU-MEDIA approach, we compare versions of presentations written in CÉU against versions of similar presentations written in traditional multimedia languages, and how these specifications are realized by the corresponding implementations. CÉU-MEDIA targets non-specialist users. Thus here we are mainly concerned with high-level multimedia languages, i.e., those with a concept of “media object” and synchronization primitives that allow for combining objects in groups and describing their behavior in time. For this reason, we choose NCL 3.0 [1] and SMIL 3 [16].

Since pure CÉU does not deal with media objects, in a first approach to compare it with NCL and SMIL we focus on the synchronization model and corresponding primitives offered by the languages. The CÉU program of Listing 1 can be viewed as a multimedia presentation if we replace the LEDs by media objects. In this case, two media objects (e.g., texts, images, audios, videos, etc.) are to be presented on screen in a loop. The first should be presented for two seconds, every two seconds, and the second should be presented for four seconds, every four seconds. At any moment, if the user presses any key, the presentation should halt.

3.1 Blinking LEDs in NCL

Listing 2 depicts the relevant parts of the multimedia version of the blinking LEDs program written in NCL. In the listing, each LED state is represented by a corresponding media object (lines 4–15). Media object Led1_on (lines 4–6) displays an image on screen for two seconds, while Led1_off (lines 7–9) displays nothing on screen for two seconds and terminates. Similarly, Led2_on (lines 10–12) displays an image on screen for four seconds, and Led2_off (lines 13–15) waits for four seconds and terminates. The duration of each object is given by the value of its explicitDur property (lines 5, 8, 11 and 14), and their presentation is interleaved by four links (lines 16–31).

When the program of Listing 2 starts, objects Led1_off and Led2_off are started (lines 2–3). These behave as countdown timers that simply wait for some time (two and four seconds, respectively) and end. When Led1_off ends, the first link (lines 16–19) is triggered and object Led1_on is started. Thus after two seconds, the first LED is displayed for two seconds, and after that the countdown timer Led1_off is restarted (lines 20–22). Similarly, when Led2_off ends, the third link (lines 25–27) is triggered and object Led2_on is started. Thus after four seconds, the second LED is displayed for four seconds, and after that Led2_off is restarted (lines 28–31). The last link (lines 32–35) establishes that when some specific key is pressed by the user the whole body (lines 1–36) is stopped, and the program terminates.

At first sight, it seems that the program of Listing 2 does what it is supposed to do: the first LED object is presented for 2s every two seconds, the second LED object is presented for 4s every four seconds, and the program terminates when the user presses a key. However, there is an issue with this program: its logical and physical behavior is simply unpredictable. The constants “2s” and “4s” are meaningless from a logical point of view. There is no guarantee that the second and fourth links (lines 20–22 and 28–31), which must be triggered exactly every 8s, will be triggered in the same time instant. In fact, in NCL, even the notion of what constitutes a “time instant” is open to interpretation. We can only hope that both are triggered as close as possible to each other. Moreover, if they happen to be triggered at exactly the same time, then there is no way to tell which of them will be executed first since link evaluation is necessarily nondeterministic.

These are logical problems in the sense that they exist independently of a particular implementation—they are caused by the ambiguous semantics of NCL and affect directly the mental model used by programmers to reason about program behavior. This loose semantics is also reflected in implementations in the form of physical dysynchrony. Even if we assume that the links are triggered at the same logical time we have no guarantee that the LEDs will appear at the same physical time on screen. Ideally, they should appear in the same video frame, but the language does not enforce that when a link is triggered, actions should be executed synchronously (at the same logical tick).

Note that the previous issues (logical meaninglessness and nondeterminism) do not occur in CÉU: programs have a synchronous, deterministic semantics, with an unambiguous notion of logical time, and count with safe concurrency primitives that are checked at compile time. What CÉU does not offer are high-level constructs for manipulating media objects; such extensions are discussed in Section 4. But before that we need to examine the SMIL version of the blinking LEDs program.

3.2 Blinking LEDs in SMIL

Listing 3 depicts the relevant parts of the blinking LEDs program written in SMIL. In the listing, each LED is represented by an image. The first image Led1_on (line 3) begins two seconds after its parent container is started (lines 2–4) and is displayed for two seconds (dur=“2s”). Similarly, the second image Led2_on (line 6) begins four seconds after its parent container is started (lines 5–7) and is displayed for
four seconds (dur="4s"). The innermost <par> containers are repeated indefinitely (repeatCount="indefinite"), and both are children of a parent <par> container (lines 1–8) that starts them in parallel as soon as the program starts and executes until key “q” is pressed by the user (end="accessKey(q")).

```
<par end="accessKey(q")>
  <par repeatCount="indefinite">
    <img id="Led1_on" begin="2s" dur="2s"/>
  </par>
  <par repeatCount="indefinite">
    <img id="Led2_on" begin="4s" dur="4s"/>
  </par>
</par>
```

Listing 3. Blinking LEDs in SMIL.

The SMIL program should behave exactly as the previous NCL program. After the program is started, Led1_on will be presented for 2s every two seconds, and Led2_on will be presented for 4s every four seconds. This situation continues until the user presses key “q”, at which point the <par> container (and consequently the whole program) terminates. Though the program of Listing 3 is conciser than its NCL version, it suffers from some semantical problems. SMIL also does not have a precise (unambiguous and well-defined) notion of logical time, so the meaning of terms such as “at the same time”, and of constants such as “2s” and “4s” is open to interpretation.

In SMIL logical time may pass even while “instantaneous” operations are being evaluated. For instance, the language does not guarantee there is no delay between subsequent repetitions of the innermost <par> containers (lines 2-4 and 5-7) of the previous program. This possibility is described in the SMIL 3.0 specification\(^\text{16}\) cf. Section “Event Sensitive”: “[The] timing of event propagation is implementation dependent, and so there are occasions in which delivery of an event may not occur because an intervening state change in the timegraph precludes event delivery.”

3.3 Synchronous languages and multimedia

The synchronous programming model was developed in the 1980s by French research groups for the trusted design of safe-critical embedded systems. The languages Esterel\(^6\), Lustre\(^10\), and Signal\(^9\) are the main products of this initial effort. Esterel is a control-oriented imperative language, while Lustre and Signal are data-oriented declarative languages—the former is a functional language and the latter is an equational language. CÉU is similar to Esterel but has a simpler semantics. The conspicuous features of all these languages is that they assume the synchronous hypothesis, i.e., that the program always reacts fast enough to external stimuli, making the actual reaction time negligible.

That this hypothesis can be maintained in real-time multimedia systems is demonstrated by the existence of specialized languages for real-time audio and video processing that implicitly assume it. (This implicit assumption is remarked by K. Barkati and P. Jouvelot\(^3\).) Examples of such languages are Pure Data\(^12\), ChucK\(^17\), CLAM\(^2\), and Faust\(^11\). ChucK (imperative) and Faust (functional) deal only with audio, while Pure Data and CLAM (both “dataflow” languages) deal with audio and video. These languages are related to CÉU-MEDIA but they target a different audience. CÉU-MEDIA targets non-specialist users whose main interest is to build a multimedia presentation consisting of synchronized media objects. In contrast, ChucK, CLAM, Pure Data, and Faust were designed with digital signal processing in mind. They target specialist users who know what their doing at the sample level and want complete control over the resulting multimedia signal.

4. CÉU-MEDIA

CÉU-MEDIA\(^4\) is a library for programming multimedia applications in CÉU. The library itself consists of three main concepts: Scene, Media, and Player. A Scene represents a top-level OS window with audio and (possibly) video output. A Media holds the description of a media object. And a Player renders a Media on a Scene. Listing 4 depicts a simple CÉU-MEDIA application that uses these concepts to present two side-by-side videos for 15s on screen, restarting them wherever both of them end.

```
var Scene s with
  this.size = Size (1080, 720);
end;
var Media m1 = Media.VIDEO ("video1.ogv", 
  Region (0,0,540,720), 1.0);
var Media m2 = Media.VIDEO ("video2.ogv", 
  Region (540,0,540,720), 1.0);
watching 15s do
  loop do
    par/and do
      await Player.play (m1, &s);
      with
        await Player.play (m2, &s);
    end
  end
end
```


Lines 1–3 define a Scene with 1080x720 pixels and store it in variable s. Lines 5–8 declare two Media descriptions, both videos. The first video (lines 5–6), variable m1, has as source “video1.ogv”; it is to be played on the region delimited by the given rectangle (Region (0,0,540,720)) with its normal volume (1.0). Similarly, the second video (lines 7–8), variable m2, has as source “video2.ogv” and is to be played on given region also with its normal volume. Note that these Media declarations are only descriptions used by players to determine what they will render on a scene. Thus at this point (line 8) nothing has happened and the screen is empty—i.e., time has not even passed.

The next statement is a watching block (lines 9–17). It defines an execution block with a duration of 15s, that is, a block that execute its body for at most 15 seconds (i.e., 15 occurrences of event “second”) and terminates. Here the body (lines 10–16) consists of an infinite loop whose sole statement is a par/and composition (lines 11–15) with two execution trails, both also consisting of a single statement (line 12 and line 14). Once executed, the par/and statement starts its trails in parallel and terminates only after both of them terminate. In this case, the first trail simply creates an anonymous player to render media m1 on scene s, starts it, and waits for its end. Similarly, the second trail creates an anonymous player to render m2 on s, starts it, and waits for its end.

When the previous program starts, the two players are created and start to render the corresponding video objects in parallel. Whenever both of them end, the whole par/and statement terminates and is immediately restarted the by the outermost loop, which means that new anonymous players are created and started. This process goes on until the 15th second is reached, at which point the watching

\Footnote{http://rodrimc.github.io/ceu-media}
block, and thus the whole program, terminates. Note that the `await` statements are the only instructions that actually block. All other instructions are conceptually instantaneous and execute in no time.

In practice, the Media is simply a structured data type, while Scene and Player are CÉU organisms: abstractions that combine data and behavior [13]. Before delving into their implementation we introduce some terminology to frame the discussion. Thinking in terms of modeling concepts and their relative level of abstraction, we regard the process of writing a multimedia application in CÉU-MEDIA as consisting of four layers, as depicted in Figure 2.

![Abstraction Levels](https://telemidia.github.io/LibPlay)

Figure 2. The abstraction layers of the authoring process.

Layer 0 is the base layer; it is simply a C API for programming multimedia. Currently, this C API is LibPlay\(^2\) a simple multimedia library based on GStreamer. Layer 1 is CÉU-MEDIA itself; it is written in CÉU upon Layer 0, hides its complexity, and exposes to the upper layer a pure high-level CÉU API (the Media type and the Scene and Player organisms). Layer 2 consists of CÉU-MEDIA programs, i.e., CÉU programs that use the CÉU-MEDIA extensions to build multimedia applications. One could stop in Layer 2, but it is possible to go further. Using CÉU mechanisms we can combine the basic abstractions of CÉU with those of CÉU-MEDIA into novel abstractions that are more suited to the description of particular scenarios. For instance, in Section 4.2, we discuss the definition of an organism for constructing multimedia slideshows. These CÉU-MEDIA extensions appear in Layer 3, the uppermost layer in terms of level of abstraction. From now on, whenever a code listing is presented, we will indicate its position in this abstraction scale.

### 4.1 Implementation

#### The Media data type

The Media type is a CÉU tagged data type. Each tag groups properties related to one of the following media types: text, image, audio, or video. A simplified version of the CÉU code that defines the Media type is presented in Listing 5.

```plaintext
var Media with
  tag MEDIA with
    var _char[256] uri; /* source uri */
    var float volume; /* sound level */
    var Region region; /* screen region */
  end
or
  tag TEXT with
    var _char[256] uri; /* source uri */
    var float volume; /* sound level */
    var Region region; /* screen region */
  end
```

Listing 5. The Media tagged data type (Layer 1).

A variable of type Media holds a set of properties but has no behavior associated to it. Although more verbose, this design promotes reuse: different Players can render the same Media instance.

#### The Scene organism

A Scene composes the output of multiple players into a synchronized multimedia scene and, under the hood, is implemented as a CÉU organism. Listing 6 depicts the interface of a Scene (lines 2–7) and its execution body (lines 8–10).

```plaintext
class Scene with
  var Size? size; /* interface */
  event mouse_click_event;
  event mouse_move_event;
  event key_event;
  event error_event;
  event (void) quit;
  do /* body */
    par/and do
      loop do
        evt = (get next event);
        emit evt;
        end
      end
      with
        every FREQ ms do
          _advance_time (FREQ * 1000000);
        end
    end
end
```

Listing 6. The Scene organism (Layer 1).

When variable of type Scene is defined, a new scene organism is created and its body starts immediately; it executes in parallel with the surrounding code until the variable goes out of scope. The Scene body performs to main tasks: (i) emits scene-level events to the application, e.g., mouse clicks, key presses and releases, errors, etc., and (ii) controls the scene clock. Every Scene maintain an internal clock to which players are synchronized. This clock only advances through explicit calls to a Layer 0 function `advance_time` (line 16, in the previous listing.) The inner workings of the scene clock and its impact on the synchronization of the output presentation are discussed in Section 4.2.

#### The Player organism

A Player renders a Media description on a Scene. Each Player is an organism that when instantiated it starts and immediately presents its associated Media on the given Scene. Later when there is no more content to be presented (i.e., the player has drained all its media content), the player stops.

```plaintext
class Player with
  var Scene $scene; /* interface */
  var Media media;
  function(Media, Scene&) => Player play;
  function(char, int) => void set_property_int;
  function(char) => int get_property_int;
  event (void) start;
  event (void) stop;
  do /* body */
    p = (allocate memory);
end
```

Listing 7. The Player organism (Layer 1).
var int now = 0;

do /* body */

var char [] & file ;

var Scene & scene ;

var Media audio = Media . AUDIO (" audio . ogg ", 1.0);

var Media vid1 = Media . VIDEO (" muted_video . ogv ", ...);

var Media vid2 = Media . VIDEO (" muted_video . ogv ", ...);

var Media vid3 = Media . VIDEO (" muted_video . ogv ", ...);

var Media vid4 = Media . VIDEO (" muted_video . ogv ", ...);

var Media audio = Media . AUDIO (" audio .ogg ", 1.0);

await 5s;

var Player p1 = Player . play (vid1 , &s);

var Player p2 = Player . play (vid2 , &s);

var Player p3 = Player . play (vid3 , &s);

var Player p4 = Player . play (vid4 , &s);

await Player . play (audio , &s);

Listing 8. Binding logical and physical time (Layer 2).

5. SAMPLE APPLICATIONS

In this section, we discuss two sample applications written in Céu-MEDIA. These applications implement simple uses cases that show that is not only feasible but also advantageous to use Céu-MEDIA when programming common multimedia synchronization scenarios. The first application (Section 5.1) is an SRT player (in fact, a Céu organism) that reads a SubRip text file and renders the corresponding subtitles. The second application (Section 5.2) is a simple multimedia slideshow that reuses the organism defined in the first application. We conclude the section (Section 5.3) with a discussion of how one could go further, from Layer 1 to Layer 2, and define an organism for slideshows which can be reused by other applications.

5.1 The SRT organism

Listing 9 depicts the partial Céu code for an SRT organism. When instantiated, the organism reads a SubRip text file and, for each subtitle entry, obtains its start time, end time, and text (lines 8–10), awaits for the amount of time corresponding to its start time (line 11), and creates a Player that renders the subtitle text for the duration of the entry.
with its value set to indefinite. Similar analogies can be made with NCL. But the crucial difference here is that the semantics of Céu is unambiguous and guarantees that the trails are, at any time, precisely and deterministically synchronized. Furthermore, in pure NCL or SMIL, it is simply impossible to create abstractions comparable to the previous SRT organism.

5.3 The Slideshow organism

The Slideshow organism captures some of the behavior of the previous slideshow application. The organism itself consists of two sets of objects: one containing media descriptions that should run in parallel, and another containing media descriptions that should be played in a sequence. When the Slideshow organism is started it creates a player for each description in these sets. Those in the parallel set are played in parallel and those in the sequence set are played in a loop, one after the other, each for a given amount of time. The organism ends when any of the players that are running in parallel terminate. Listing 11 depicts the Céu-MEDIA code of this organism.

Listing 11. The Slideshow organism (Layer 2).

In Listing 11 the parallel and sequence sets are represented by the media lists (lines 3–4) in the organism interface. The interface also has variables that determine target scene (scene, line 2), the duration of each entry in the sequence set (time, line 5), and the specific key which causes the organism to terminate (quit, line 6). The organism body consists of two parallel trails in a par/or composition. The first trail (line 7) simply waits for the given quit key before terminating, while the second trail (lines 11–30) implements the slideshow semantics, that is, traverses the media lists recursively (via traverse statements) creating the players and waiting for the appropriate events, e.g., time seconds before stopping each player created lines 22–24.

Listing 12 depicts a Céu-MEDIA program that uses the previous Slideshow. The program simply creates the scene, the media lists, and the organism.
var Scene scene with this.size = Size (800, 585); end;
pool MediaList parallel = {
  new MediaList.CONS (Media.AUDIO ("piano.ogg", .5)),
  new MediaList.CONS (Media.IMAGE ("frame.png", ...)),
  new MediaList.CONS (Media.IMAGE ("img1.jpg", ...)),
  new MediaList.CONS (Media.IMAGE ("img2.jpg", ...)),
  new MediaList.CONS (Media.IMAGE ("img3.jpg", ...)),
  new MediaList.CONS (Media.NIL ())),
pool MediaList() sequence = {
  new MediaList.CONS (Media.AUDIO ("piano.ogg", .5))
};
do Slideshow with
  this.scene = &scene;
  this.parallel = &parallel;
  this.sequence = &sequence;
  this.time = 10;
  this.quit = 'q';
end;

Listing 12. A program that uses the Slideshow organism (Layer 2). Alternatively, we can specify the previous program using a Lua table, since CÉU can be seamlessly integrated with Lua. The Lua version is depicted in Listing 13. Both versions, Listing 12 and 13, are equivalent, i.e., they produce exactly the same resulting presentation. Here we chose Lua for more convenience. Any higher-level syntax could be used, provided that there is a corresponding CÉU code to parse it. Finally, note that this example illustrates that from a small set of abstractions exposed by CÉU-MEDIA it is possible to create higher-level constructs targeting non-specialist users. Such usage resembles the usage of template languages such as TAL [15] or XTemplate [7] in the domain of XML languages.

rect = {76,74,650,440};
SLIDESHOW = {
  width = 800, height = 585,
  background = {
    {tag='audio', uri='piano.ogg', volume=.5},
    {tag='image', uri='frame.png', rect={0,0,800,585}},
  },
  sequence = {
    {tag='image', uri='img1.jpg', rect=rect},
    {tag='image', uri='img2.jpg', rect=rect},
    {tag='image', uri='img3.jpg', rect=rect},
  }
};

Listing 13. A Lua version of the slideshow program (Layer 3).

6. CONCLUSION

In this paper, we investigated the use of the synchronous language CÉU for programming multimedia applications, in particular, those applications that can be described as a set of synchronized media objects. The concrete result of this investigation is CÉU-MEDIA, a library for multimedia programming in CÉU. The programming model and abstractions offered by CÉU-MEDIA are similar to that of the traditional high-level multimedia languages NCL and SMIL, but avoids their inflexibility, ambiguity, and synchronization problems. This is only possible because CÉU-MEDIA takes full advantage of CÉU features: its integration with C, its abstraction mechanisms (tagged types and organisms), and its semantics—which is unambiguous, deterministic, and allows for a precise control of time. And because, in the implementation of CÉU-MEDIA, we took care to ensure that the properties of the semantics of CÉU are reflected in the output multimedia presentation.

On the theory side, this work is another evidence that the synchronous approach might be an adequate solution to the longstanding semantical problems of NCL and SMIL, and possibly HTML. In fact, an approach to these problems, and possible future work, is to investigate how CÉU and CÉU-MEDIA can be used to implement a NCL or SMIL player—which would indirectly “solve” problem of ambiguity in their specification.

Other future work include improving the current implementation of CÉU-MEDIA. For instance, in the current implementation some rendering flaws may be noticed as the skew between the presentation time and the physical time increases (specially for sounds due their high sampling frequency). We are investigating solutions to minimize this problem, and to extend the implementation, by adding operations to pause and seek in players, and by investigating the problem of program fast-forwarding and rewinding. Finally, another possibility is extending the CÉU-MEDIA model to deal with distributed applications, where communication latency makes the synchronous hypothesis unfeasible.

REFERENCES