CÉU-MEDIA: Local Inter-Media Synchronization using CÉU

Rodrigo C. M. Santos[1] Guilherme F. Lima[1] Francisco Sant'Anna[2] Noemi Rodriguez[1*]

[1]Department of Informatics
PUC-Rio, Rio de Janeiro, Brazil
{rsantos,glima,noemi}@inf.puc-rio.br

[2]Department of Informatics and Computer Science
UERJ, Rio de Janeiro, Brazil
francisco.santanna@gmail.com

ABSTRACT

The semantics of current multimedia languages is informal and may lead to the development of ambiguous applications. In this paper we investigate the use of the synchronous language Céu for programming local multimedia applications, in particular, those applications that can be described as a set of synchronized media objects. Céu has a deterministic, concise and accurate semantics which, along with high-level programming constructs, makes the language an attractive alternative for developing multimedia applications. We also present Céu-Media, a library for programming multimedia in Céu. Céu-Media implementation guarantees that the properties of the Céu synchronous semantics are reflected in the multimedia presentation, ensuring inter-media synchronization. We compare the synchronization paradigm of Céu with those of NCL and SMIL, and examine the implementation of 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

Current multimedia languages (e.g., NCL, SMIL, HTML) have an informal semantics which is described in verbose manuals and written in natural language. The lack of rigor often leads to the development of ambiguous or inconsistent applications. This issue has motivated the development of formal methods to validate applications specified in these languages [10, 14, 7]. However, most of these works are concerned with static validation; their are complex and impractical for real-time performance, and usually do not cover the whole semantics of the target languages [9].

In this paper, we approach the problem of multimedia authoring by exploring the use of the synchronous language Céu. Céu has some characteristics that make it attractive for specifying multimedia applications. The first one is flexibility. In addition to the traditional programming language control constructs (e.g., loops, conditional expressions, functions), Céu has high-level statements (e.g., parallel compositions, awaits) and abstractions (e.g., organisms) that allow one to create further abstractions more suitable to particular scenarios.

In second place, Céu has a straightforward, accurate semantics induced by the synchronous hypothesis and enforced determinism. In Céu, programs react to external events. 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.

These advantages contrast with the use of NCL or SMIL (or similar languages) for specifying multimedia applications. In both languages, any extension must be done externally via pre-processors or via scripts (Lua or JavaScript) that modify the original program; they have ambiguous and inconsistent semantics, and they do not allow the precise control of the presentation output neither at specification nor at implementation level.

Our investigation led to the development of Céu-Media, a library for authoring multimedia applications using Céu. With Céu-Media authors describe multimedia presentations in an abstraction level close to that adopted by traditional high-level multimedia languages, while taking advantage of Céu features. In its implementation, we strove to maintain the accuracy imposed by the synchronous semantics of Céu in the audiovisual output presentation.

The rest of the paper is organized as follows. In Section 2, we briefly present 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.

*The authors would like to thank the Brazilian National Counsel of Technological and Scientific Development (CNPq) for funding this research.

WebMedia '16, November 08-11, 2016, Teresina, PI, Brazil

DOI: http://dx.doi.org/10.1145/2976796.2976856

2. CÉU IN A NUTSHELL

Céu [17] is a synchronous programming language for developing safe concurrent programs. By *synchronous*, we mean that its programs assume the synchronous hypothesis [5], 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 black-box 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. Originally the language was designed to be an alternative for developing control-intensive embedded applications (Wireless Sensor Network domain), therefore Céu runtime is resource-efficient and maintains real-time responsiveness even in constrained devices [17].

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 access to shared variables is always consistent [17].

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.

```
1  input void KEY;
2  par/or
3  do                  /* trail 1 */
4    loop do
5      await 2s;
6      _Led1_on()
7      await 2s;
8      _Led1_off()
9    end
10 with                /* trail 2 */
11   loop do
12     await 4s;
13     _Led2_on()
14     await 4s;
15     _Led2_off()
16   end
17 with                /* trail 3 */
18   await KEY;
19 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` on, awaits 2 more seconds and turns it off. 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 the 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 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 or terminate, 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 E occurs, the environment gives control back to the program; all trails that are blocked waiting for event E 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-seconds timeline repeats indefinitely until some key is pressed. 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.

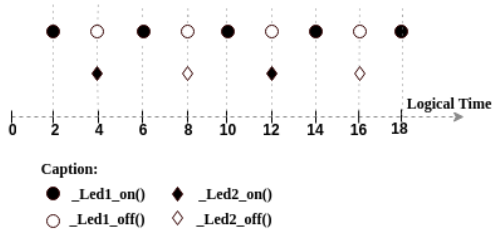

Figure 1. Timeline of the blinking LEDs program.

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.

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 [12], and Signal [11] 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 common 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 hard real-time multimedia systems is demonstrated by the existence of specialized languages for audio and video processing that implicitly assume it [3]. Examples of such languages are Pure Data [15], ChucK [20], CLAM [2], and Faust [13]. ChucK (imperative) and Faust (functional) deal only with audio, while Pure Data and CLAM (both "dataflow" languages) deal with audio and video. In this paper, we also argue for the adoption of the synchronous hypothesis in the multimedia domain. But our main concern is the accurate specification and maintenance of inter-media synchronization (most of the mentioned languages were designed with digital signal processing in mind) and we regard multimedia applications as soft-real time systems.

3. COMPARING CÉU TO NCL AND SMIL

Céu-Media (discussed in next section) 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 [19].

Because pure Céu does not deal with media objects, we first compare it with NCL and SMIL as regards to the synchronization model and corresponding primitives offered by the languages. One may view the Céu program of Listing 1 as a multimedia presentation if we replace the LEDs by media objects. In this case, the program presents two media objects (e.g., texts, images, audios, videos, etc.) 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 `explicityDur` 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 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 20–23). Similarly, when `Led2_off` ends, the third link (lines 24–27) 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) stops, the program terminates.

```
1  <body id="blink">
2    <port id="pLed1_off" component="Led1_off"/>
3    <port id="pLed2_off" component="Led2_off"/>
4    <media id="Led1_on" src="Led1.png">
5      <property name="explicitDur" value="2s"/>
6    </media>
7    <media id="Led1_off">
8      <property name="explicitDur" value="2s"/>
9    </media>
10   <media id="Led2_on" src="Led2.png">
11     <property name="explicitDur" value="4s"/>
12   </media>
13   <media id="Led2_off">
14     <property name="explicitDur" value="4s"/>
15   </media>
16   <link xconnector="onEndStart">
17     <bind role="onEnd" component="Led1_off"/>
18     <bind role="start" component="Led1_on"/>
19   </link>
20   <link xconnector="onEndStart">
21     <bind role="onEnd" component="Led1_on"/>
22     <bind role="start" component="Led1_off"/>
23   </link>
24   <link xconnector="onEndStart">
25     <bind role="onEnd" component="Led2_off"/>
26     <bind role="start" component="Led2_on"/>
27   </link>
28   <link xconnector="onEndStart">
29     <bind role="onEnd" component="Led2_on"/>
30     <bind role="start" component="Led2_off"/>
31   </link>
32   <link xconnector="onKeySelectionStop">
33     <bind role="onKeySelection" component="blink"/>
34     <bind role="stop" component="blink"/>
35   </link>
36 </body>
```

Listing 2. Blinking LEDs in NCL.

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 20–23 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 nondeterministic.

These problems are caused by the ambiguous semantics of NCL and they exist independently of a particular implementation. This loose semantics is also reflected in implementations in the form of physical dyssynchrony. 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). In our tests, the Ginga-NCL reference implementation did not maintain the images blinking in-sync.

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 `begin` attributes emulate the countdowns `Led1_off` and `Led2_off` of the NCL counterpart. 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 (`end="accessKey(q)"`).

```
1 <par end="accessKey(q)">
2  <par repeatCount="indefinite">
3   <img id="Led1_on" begin="2s" dur="2s" src="L1.png"/>
4  </par>
5  <par repeatCount="indefinite">
6   <img id="Led2_on" begin="4s" dur="4s" src="L2.png"/>
7  </par>
8 </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 seconds 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 same 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 [19, 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". In our tests using the SMIL Ambulant Player, we could not notice delays between the generation and the processing of events.

4. CÉU-MEDIA

Céu-Media[1] is a library for programming multimedia applications in Céu. The library itself has three main concepts: `Scene`, `Media`, and `Player`. A `Scene` represents a top-level OS window with audio and 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 whenever both end.

```
1  var Scene s with
2    this.size = Size (1080, 720);
3  end;
4  var Media m1 = Media.VIDEO ("video1.ogv",
5                      Region(0, 0, 540, 720), 1.0);
6  var Media m2 = Media.VIDEO ("video2.ogv",
7                      Region(540, 0, 540, 720), 1.0);
8  watching 15s do
9    loop do
10     par/and do
11       await Player.play (m1, &s);
12     with
13       await Player.play (m2, &s);
14     end
15   end
16 end
```

Listing 4. Two side-by-side videos in Céu-Media.

Lines 1–3 define a `Scene` with 1080x720 pixels and store it in variable `s`. Lines 4–7 declare two `Media` descriptions, both videos. The first video (lines 4–5), 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 6–7), variable `m2`, has as source "video2.ogv" and is to be played on the given region (`Region (540,0,540,720)`) 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 7) nothing has happened and the screen is empty—in fact, time has not even passed.

The next statement is a `watching` block (lines 8–16). It defines an execution block with a duration of 15s, that is, a block that executes its body for at most 15 seconds, i.e., 15 occurrences of event "second", and terminates. Here the body (lines 9–15) consists of an infinite loop whose sole statement is a `par/and` composition (lines 10–14) with two execution trails, each also consisting of a single statement (line 11 and line 13). 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 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 program in Listing 4 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 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` 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 a structured data type, while `Scene` and `Player` are Céu organisms: abstractions that combine

[1] http://rodrimc.github.io/ceu-media

data and behavior [16]. Before detailing their implementation we introduce some terminology. 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, depicted in Figure 2.

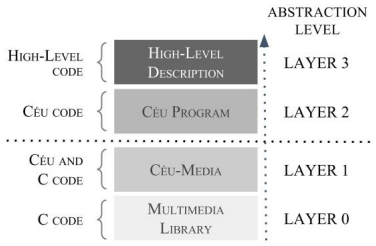

Figure 2. The abstraction layers of the authoring process.

Layer 0 is the base layer; it is 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 better suited to the description of particular scenarios. For instance, in Section 5 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.

```
1  data Media with
2    tag VIDEO with
3      var _char[255] uri;  /* source uri */
4      var Region region;   /* screen region */
5      var float volume;    /* sound level */
6    end
7  or
8    tag IMAGE with
9      var _char[255] uri;  /* source uri */
10     var Region region;   /* screen region */
11   end
12 or
13   tag AUDIO with
14     var _char[255] uri;  /* source uri */
15     var float volume;    /* sound level */
16   end
17 or
18   tag TEXT with
19     var _char[255] text; /* text to render */
20     var uint color;      /* text color */
21     var Region region;   /* screen region */
22   end
23 end
```

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

[2]https://github.com/TeleMidia/LibPlay

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` description.

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 2–4) and a simplified version of its execution body (lines 5–16).

```
1  class Scene with
2    var Size? size;            /* interface */
3    event mouse_click_event;
4    event key_event;
5  do                           /* body */
6    par/and do
7      loop do
8        evt = ⟨get next event⟩;
9        emit evt;
10     end
11   with
12     every FREQ ms do
13       _advance_time (FREQ * 1000000);
14     end
15   end
16 end
```

Listing 6. The `Scene` organism (Layer 1).

When a 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 two main tasks: (i) it emits scene-level events to the application, e.g., mouse clicks, key presses and releases, etc., and (ii) it controls the scene clock. Every `Scene` maintains an internal clock to which players are synchronized. This clock only advances through explicit calls to a Layer 0 function `advance_time` (line 13, 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, immediately presents its associated `Media` on the given `Scene`. When there is no more content to be presented (i.e., the player has drained all of its media content), the player stops (the organism ends).

```
1  class Player with
2    var Scene &scene;          /* interface */
3    var Media media;
4    function(Media, Scene&) => Player play;
5    function(char, int) => void set_property_int;
6    function(char) => int get_property_int;
7    event (void) start;
8    event (void) stop;
9  do                           /* body */
10   p = ⟨allocate memory⟩;
11   finalize
12     _start (p);
13   with
14     _stop (p);
15   end
16   await p;
17 end
```

Listing 7. The `Player` organism (Layer 1).

Listing 7 depicts a sketch of the Céu code that defines the `Player` organism. The `Player` interface consists of its data (associated `Media` and `Scene`, lines 2–3), exposed functions

(constructor plus property getters and setters, lines 4–6), and events (start and stop, lines 7–8). The player constructor (function `play`) takes a `Media` and a `Scene` and returns a new `Player`, and the getters and setters are used to get or set player properties, which control the audiovisual characteristics of the player output. In Listing 7, only the functions for getting and setting integer properties are shown, namely, `get_property_int` and `set_property_int`; there are similar functions for the other primitive data types.

Starting the presentation of a `Media` might take a non-negligible time—because it involves complex operations such as resolving the content URI, opening the content file, decoding it, transforming the raw samples, etc. So the `Player` uses an asynchronous start process: it loads the Level 0 player, requests an asynchronous start, waits for its completion, and emits a corresponding (Level 1) `start` event. Similarly, when the Level 0 player notifies that its samples have been exhausted, the `Player` emits a corresponding (Level 1) `stop` event. From the logical point of view, a `Player` starts at the moment (logical time) its constructor is called—it uses the `start` event to notify the completion of the asynchronous start. To reflect what is specified in the source code, the `Scene` has to consider the moment players have been created, and not the moment their `start` event is emitted. Thus, for timed media, initial content may not be rendered if the asynchronous start takes too long to complete.

In Céu, the organism body may have a `finalize` block that executes a given piece of code whenever the organism is killed or finishes [17] (such blocks are similar to destructor methods). In the Listing 7, we use a finalize block (lines 11–15) to guarantee that the Level 0 player is stopped whenever the corresponding `Player` variable goes out of scope. This ensures not only that the player is stopped, but also that the allocated resources are properly released.

4.2 Synchronization

Every `Scene` has an internal monotonic clock that starts with 0 and advances only through explicit calls to the Layer 0 function `advance_time()`. Such calls are triggered by the scene organism itself (i.e., Céu-Media users should not worry about calling this function). For instance, in Listing 6, the `Scene` advances its clock every `FREQ` milliseconds (lines 12–14), where `FREQ` is an internal constant, by the corresponding amount of time. This call binds the logical time events of Céu with the "physical" clock used to synchronize all players in a given scene—or more precisely, to time-stamp the samples produced by these players.

To illustrate the consequence of this binding of logical and physical time, consider the program depicted in Listing 8. The program creates a scene (lines 1–3), four muted videos (with no audio tracks), `vid1`, `vid2`, `vid3`, `vid4` (lines 4–7), and an audio (line 8), `audio`. Then it waits for five seconds (line 9) and creates four players (lines 10–13), `p1`, `p2`, `p3`, and `p4`, initializing each with one of the previous video media; these are started as soon as they are created. Finally, it creates an anonymous player (line 14) to play the audio media, starts it, and waits for its end (`stop` event).

The only instructions that actually take time in this program are the `await` statements in lines 9 and 14, and the code that advances the scene clock (Listing 6, lines 12–14)—and they all consume exactly the specified amount of logical time. This means that logical time does not pass while the players are being created and started. Moreover, because the logical clock drives the physical (scene) clock, this also means that no samples are timestamped with distinct values during this time. Because the physical time actually passes while the program creates the players, without this "deterministic" control over the scene clock, each `Player` would set a different timestamp value on the produced samples. This would happen even though they have been created in the same reaction. The program in Listing 8 produces a presentation that renders the four videos and their respective audio in-sync.

```
1  var Scene s with
2    this.size = Size (1080, 720);
3  end;
4  var Media vid1 = Media.VIDEO ("muted_video.ogv", ...);
5  var Media vid2 = Media.VIDEO ("muted_video.ogv", ...);
6  var Media vid3 = Media.VIDEO ("muted_video.ogv", ...);
7  var Media vid4 = Media.VIDEO ("muted_video.ogv", ...);
8  var Media audio = Media.AUDIO ("audio.ogg", 1.0);
9  await 5s;
10 var Player p1 = Player.play(vid1, &s);
11 var Player p2 = Player.play(vid2, &s);
12 var Player p3 = Player.play(vid3, &s);
13 var Player p4 = Player.play(vid4, &s);
14 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 use 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 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 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.

```
1  class SRT with                    /* interface */
2    var Scene &scene;
3    var char[] &file;
4    var int y_offset;
5  do                                /* body */
6    var int now = 0;
7    loop entry in ⟨subtitle entry in file⟩ do
8      var int from = get_start_time (entry);
9      var int to = get_end_time (entry);
10     var char[]text = get_subtitle_text (entry);
11     await (from - now)ms;
12     watching (to - from)ms do
13       var Media text = Media.TEXT (text, 0xffff0000,
14                        Region(0, y_offset, 800, 100));
15       await Player.play(text, &scene);
16     end
17     now = to;
18   end
19 end
```

Listing 9. The `SRT` organism (Layers 1–2).

The complete code of the `SRT` organism demands the use of asynchronous I/O operations for reading the SRT file, along

with `await` statements for synchronizing the asynchronous calls, as the use of traditional blocking I/O would violate the synchronous hypothesis. Thus a programmer writing this organism needs to work on Layers 1 (asynchronous I/O) and 2 (text rendering via Céu-Media). Finally, note that this application cannot be directly implemented in NCL, SMIL, or HTML without resorting to external scripts.

5.2 A multimedia slideshow

The slideshow we consider consists of three images. Each one is presented for five seconds while a piano soundtrack is played in background (in a loop) and synchronized subtitles are shown over the images. The slideshow terminates when all three images are displayed or when there are no more subtitles to be presented or any key is pressed. Listing 10 depicts the Céu-Media code of this application.

```
1  var Scene s with this.size = Size (800, 585); end;
2  var Media piano = Media.AUDIO ("piano.ogg", .5);
3  var Media img1 = Media.IMAGE ("img1.jpg", ...);
4  var Media img2 = Media.IMAGE ("img2.jpg", ...);
5  var Media img3 = Media.IMAGE ("img3.jpg", ...);
6  par/or do
7    loop do await Player.play (piano, &s); end
8  with
9      watching 5s do await Player.play (img1, &s); end
10     watching 5s do await Player.play (img2, &s); end
11     watching 5s do await Player.play (img3, &s); end
12 with
13   await SRT (&s, "subtitle.srt", 485);
14 with
15   await s.key_event;
16 end
```

Listing 10. A multimedia slideshow (Layer 2).

Listing 10 begins creating the scene and the necessary media descriptions (lines 1–5). Then it starts four execution trails in a `par/or` composition—the composition, and thus the program, ends when any of these trails end. The first trail (line 7) creates an anonymous player to render the background piano music in a loop (every time the player ends it is recreated and restarts the music). The second trail (lines 9-11) presents the three images (in corresponding players) each for five seconds. The third trail (line 13) creates an `SRT` organism to present the subtitles and waits for it to finish before terminating. Finally, the fourth trail (line 15) awaits for a scene `key_event` before terminating.

The previous `par/or` composition (lines 6–16) and the sequence of `watching` statements (lines 9–11) resemble the `par` (with its `endsync` attribute equals to `first`) and `seq` SMIL containers. The `watching` blocks resemble SMIL's `dur` attribute, while the counterpart of the previous `loop` statement is the `repeatCount` attribute of SMIL, 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 impossible to create abstractions comparable to the previous `SRT` organism

5.3 The Slideshow organism

It is possible using Céu constructs to declaratively describe a slideshow application. For instance, Listing 11 depicts a Céu program that defines into two `MediaList`, `parallel` in lines 2–5 and `sequence` in lines 6–10, media objects that should be presented in parallel (background) and in sequence. This program uses the Céu organism `Slideshow` (Listing 12) to actually present these objects.

```
1  var Scene scene with this.size = Size (800, 585); end;
2  pool MediaList[] parallel =
3    new MediaList.CONS (Media.AUDIO ("piano.ogg", .5),
4        MediaList.CONS (Media.IMAGE ("frame.png", ...),
5        MediaList.NIL ()));
6  pool MediaList[] sequence =
7    new MediaList.CONS (Media.IMAGE("img1.jpg", ...),
8        MediaList.CONS (Media.IMAGE("img2.jpg", ...),
9        MediaList.CONS (Media.IMAGE("img3.jpg", ...),
10       MediaList.NIL ())));
11 do Slideshow with
12     this.scene    = &scene;
13     this.parallel = &parallel;
14     this.sequence = &sequence;
15     this.time     = 10;
16     this.quit     = 'q';
17 end;
```

Listing 11. A declarative description of a slideshow in Céu (Layer 2).

Despite the imperative nature of Céu, this code contains basically declarations of media lists and an instantiation of an organism (lines 11–17). The `Slideshow` organism (Listing 12) actually implements the application logic, by capturing some of the behavior of the previous slideshow program. 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 (the `spawn` statement in line 14 spawns instances of `Player`s) and those in the `sequence` set are played in a loop, one after the other, each for a given amount of time (the `await` statement in line 24 creates a `Player` and waits until it finishes). The organism ends when the key `quit` (set when the organism is instantiated) is pressed or all media within the `sequence` list terminate. Listing 12 depicts the Céu-Media code of this organism.

```
1  class Slideshow with        /* interface */
2    var Scene &scene;
3    pool MediaList[] &parallel;
4    pool MediaList[] &sequence;
5    var uint time;
6    var char quit;
7  do                          /* body */
8    par/or do
9      key = await s.key_event until (key == quit);
10   with
11     traverse list in && this.parallel do
12       watching *list do
13         if list:CONS then
14           spawn Player.play (list:CONS.media, &s);
15           traverse &&list:CONS.next;
16         end
17       end
18     end
19     loop do
20       traverse list in && this.sequence do
21         watching *list do
22           if list:CONS then
23             watching (time)s do
24               await Player.play (list:CONS.media, &s);
25             end
26             traverse &&list:CONS.next;
27           end
28         end
29       end
30     end
31   end
32 end
```

Listing 12. The Slideshow organism (Layer 2).

In Listing 12, 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 the

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 9) 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 in line 24.

Alternatively, we can specify a slideshow program using a Lua table (Céu can be seamless integrated with Lua). The Lua version is depicted in Listing 13. Both versions, Listing 11 and 13, are equivalent, i.e., they produce exactly the same resulting presentation. Here we chose Lua for mere 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 nonspecialist users. Such usage resemble the use of template languages such as TAL [18] or XTemplate [8] in the domain of XML languages.

```
1  rect = {76,74,650,440}
2  SLIDESHOW = {
3    width = 800, height = 585,
4    background = {
5      {tag='audio', uri='piano.ogg', volume=.5},
6      {tag='image', uri='frame.png', rect={0,0,800,585}},
7  },sequence = {
8      {tag='image', uri='img1.jpg', rect=rect},
9      {tag='image', uri='img2.jpg', rect=rect},
10     {tag='image', uri='img3.jpg', rect=rect},
11 },}
```

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

6. CONCLUSION

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 can be used to implement an NCL or SMIL player—which would indirectly "solve" the problem of ambiguity in their specification.

Even though Céu has a different target audience than traditional declarative multimedia languages, we have illustrated in this paper how one can use the constructs and features of Céu to create abstractions suitable to users of those languages. The imperative nature of Céu, however, may be an issue for authors used to the declarative paradigm. An approach to overcome this problem is to explore the development of higher-level abstractions, as illustrated by the use of Lua tables for creating slideshows.

The enslaving of the presentation clock to the logical clock of Céu programs leads to some rendering flaws that become more noticeable 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.

Finally, our research has indicated that Céu is well-suited for developing local multimedia applications. As the synchrony hypothesis cannot be assumed in distributed settings due to communication latency, further research must be conducted to understand the limits of synchronous languages in developing distributed multimedia applications. We are investigating how to implement different architectures to support the requirements of these systems.

REFERENCES

[1] ABNT NBR 15606-2. *Digital Terrestrial TV — Data Coding and Transmission Specification for Digital Broadcasting — Part 2: Ginga-NCL for Fixed and Mobile Receivers: XML Application Language for Application Coding*. ABNT, São Paulo, SP, Brazil, February 2016.

[2] X. Amatriain, P. Arumi, and D. Garcia. A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, 14(1):15–32, 2008.

[3] K. Barkati and P. Jouvelot. Synchronous programming in audio processing: A lookup table oscillator case study. *ACM Computing Surveys*, 46(2):24:1–24:35, December 2013.

[4] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. MIT Press, 2000.

[7] J. A. dos Santos, C. Braga, and D. C. Muchaluat-Saade. Automating the analysis of NCL documents with a model-driven approach. In *WebMedia 2013*, pages 193–200, New York, New York, USA, nov 2013. ACM Press.

[8] J. A. F. dos Santos and D. C. Muchaluat-Saade. XTemplate 3.0: spatio-temporal semantics and structure reuse for hypermedia compositions. *Multimedia Tools and Applications*, 61(3):645–673, jan 2011.

[9] G. F. Lima. *A synchronous virtual machine for multimedia presentations*. PhD thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2015.

[10] A. Ghomari, N. Belheziel, F. Z. Mekahlia, and C. Djeraba. *Towards a formal approach for verifying temporal coherence in a SMIL document presentation*, volume 8216 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[11] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Circuits, Systems, and Computers*, 2003.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] Y. Orlarey, D. Fober, and S. Letz. FAUST: An efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music*. 2009.

[14] D. Picinin Júnior, C. Koliver, C. A. S. Santos, and J.-M. Farines. Verifying Hypermedia Applications by Using an MDE Approach. *System Analysis and Modeling: Models and Reusability*, 8769:174–189, 2014.

[15] M. S. Puckette. *The Theory and Technique of Electronic Music*. World Scientific Publishing Company, Singapore, 2007.

[16] F. Sant' Anna, R. Ierusalimschy, and N. Rodriguez. Structured synchronous reactive programming with Céu. In *Proc. of the 14th International Conference on Modularity*, pages 29–40, New York, New York, USA, 2015. ACM Press.

[17] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-Level Concurrency on Resource-Constrained Nodes. In *SenSys '13*, New York, New York, USA, nov 2013. ACM Press.

[18] C. d. S. Soares Neto, L. F. G. Soares, and C. S. de Souza. TAL—Template Authoring Language. *Journal of the Brazilian Computer Society*, 18(3):185–199, sep 2012.

[19] W3C Recommendation 01 December 2008. *Synchronized Multimedia Integration Language (SMIL 3.0)*. World Wide Web Consortium (W3C), December 2008.

[20] G. Wang and P. Cook. ChucK: A Programming Language for On-the-fly, Real-time Audio Synthesis and Multimedia. In *Proc. of 12th ACM Multimedia*, pages 812–815, 2004.