gingaf: Headless and Multiplatform Execution of ITU-SBTVD NCL Applications

Alan L.V. Guedes
Computer Science, University of Reading
Reading, UK
a.guedes@reading.ac.uk

Abstract

The Nested Context Language (NCL) is a core component of the ITU-T and SBTVD television standards. It is legally mandated and installed on 100% of certified Digital TV sets and set-top boxes across Brazil and several Latin American countries. This massive deployment impacts over 60 million people. However, the ecosystem of NCL development tools is limited. In this context, we present `gingaf`, an open-source implementation of an NCL player built in Flutter. It supports a headless execution engine that schedules NCL events in-memory without requiring any UI and provides a foundation for NCL developer tooling. We also propose a UI component that achieves multiplatform visual rendering of NCL documents on Windows, mobile, and Web from a single codebase.

CCS Concepts

• **Information systems** → **Multimedia content creation**.

Keywords

TV, NCL, Ginga, Declarative Multimedia

ACM Reference Format:
Alan L.V. Guedes. 2026. `gingaf`: Headless and Multiplatform Execution of ITU-SBTVD NCL Applications. In *Proceedings of the 34th ACM International Conference on Multimedia (MM '26), November 10–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3767308.3834744

1 Introduction

The Nested Context Language (NCL) is a core component of the ITU-T [5] and SBTVD [1] television standards. NCL defines a rich temporal and spatial synchronization model, allowing authors to specify how media objects interact over time using links and connectors. The NCL player is called Ginga. Ginga implementations are legally mandated and installed on 100% of certified Digital TV sets and set-top boxes sold in Brazil and several other Latin American countries using the ISDB-T international television standard. Because it is legally mandated by local telecommunications entities (such as the SBTVD in Brazil), every major manufacturer (including Samsung, LG, Sony, and TCL) must include it directly into their television hardware. This wide adoption results in over 60 million people impacted. Given this reachability, NCL is also used by the government, e.g., Brazil4D [2] app targets low-income families to access public services, health information, and educational content. Recently, the application coding specification for the next-generation Brazilian digital television standard (aka, TV 3.0) has continued to advance the use of NCL [4].

Despite the above context, NCL developers and researchers still face two main development barriers. **P1) Lack of Accessible NCL Development Environment**: Traditional NCL players, such as the reference PUC-Rio C++ Ginga implementations, are tightly coupled to specific platform architectures (targeted Linux-based digital TV receivers), some require specialized OS setups to compile and run. This creates a steep barrier to entry that contrasts sharply with the trend in modern programming ecosystems, where major languages provide instant web-based playgrounds: HTML is directly executable in any browser[1], and even native compiled languages such as Go[2] offer zero-install online environments for immediate experimentation. **P2) Lack of NCL Debugging Features**: Current NCL players do not allow developers to pause execution to investigate intermediate states or run the synchronization engine headless (without the graphical user interface), making testing and debugging highly manual and time-consuming.

`gingaf` (see available software[3] and demonstrations [4]) addresses these challenges by introducing a modern, decoupled, and portable implementation of the NCL ecosystem. Written in Dart and built on the Flutter framework, `gingaf` separates document execution logic from the UI presentation component, exposing a clean API between them. This decoupling yields two primary contributions:

- **Multiplatform execution (targeting P1)**: by leveraging Flutter, `gingaf` offers portable presentation support across diverse operating systems (Windows, macOS, Linux, iOS, Android, and Web) from a single codebase, removing the specialized hardware and environment setup barriers.
- **Headless execution (targeting P2)**: by separating the document state machine from the NCL UI component, the execution engine allows developers to run without a UI, advance time deterministically via a virtual clock, and tick calls to inspect and debug NCL application states in memory.

The remaining sections are organized as follows: NCL background (which can be skipped by readers familiar with NCL); related work; `gingaf` architecture; results and reproducibility instructions; and final remarks.

[1] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/img
[2] https://go.dev/play/
[3] https://github.com/ginga-org-br/gingaf/releases/tag/v0.1.0
[4] https://alanlivio.github.io/paper-mm26-gingaf/

Unpublished working draft. Not for distribution.

MM '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 34th ACM International Conference on Multimedia (MM '26), November 10–14, 2026, Rio de Janeiro, Brazil*, https://doi.org/10.1145/3767308.3834744.

2 NCL Background

NCL acts as a glue language, defining temporal and spatial synchronization among diverse media assets and viewer interactions, without prescribing media content formats [7]. A `<media>` element may be an image, video, audio, text, or even imperative (e.g., Lua, Java) and declarative (e.g., HTML, SMIL) objects. It is defined by its type, content location, and properties. Furthermore, a media object can define an explicit duration (`explicitDur`) or temporal segments via the `<area>` element. The `<link>` element defines temporal relationships between nodes. It has a set of conditions that need to be satisfied to trigger a set of actions. The conditions can test the occurrence or property changes of media nodes or their segments; actions change node occurrences (start, pause, resume, stop) and properties (set).

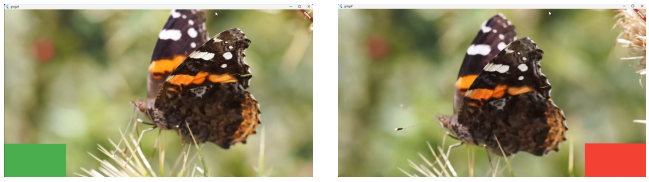
To demonstrate the structure of NCL applications, we use a sample application (`video.ncl`) presented in Listing 1. The presentation defines a primary video (`video`) in lines 4–6, a green background media (`green_bg`) in lines 7–10, and a red background media (`red_bg`) in lines 11–15 with an explicit duration of 2 seconds. Temporal behavior is coordinated by link elements: link `l1` starts the green background when area `a1` of the video begins (at 2s), and link `l2` stops the green background and starts the red background when area `a1` ends (at 4s). The application ends when the `red_bg` ends at 6s (explicitDur 2s) and the main video ends decoding at 7s (video content duration). Visual execution screenshots at these timestamps are shown in Figure 1a and Figure 1b. Video demonstrations are also present online[5].

```
1  <ncl>
2    <body>
3      <port id="init" component="video"/>
4      <media id="video" src="https://flutter.github.io/assets-for-
           api-docs/assets/videos/butterfly.mp4">
5        <area id="a1" begin="2s" end="4s"/>
6      </media>
7      <media id="green_bg">
8        <property name="background" value="green"/>
9        <property name="bounds" value="0%,80%,20%,20%"/>
10     </media>
11     <media id="red_bg">
12       <property name="background" value="red"/>
13       <property name="bounds" value="80%,80%,20%,20%"/>
14       <property name="explicitDur" value="2s"/>
15     </media>
16     <link id="l1">
17       <bind role="onBegin" component="video" interface="a1"/>
18       <bind role="start" component="green_bg"/>
19     </link>
20     <link id="l2">
21       <bind role="onEnd" component="video" interface="a1"/>
22       <bind role="stop" component="green_bg"/>
23       <bind role="start" component="red_bg"/>
24     </link>
25   </body>
26 </ncl>
```

Listing 1: video.ncl source code

Other advanced NCL features (not discussed for simplicity) include support for embedding scripts written in the lightweight scripting language Lua to implement complex interactive behavior, and a system of NCL editing commands that allows modifying applications on-the-fly. For example, in the application defined in Listing 1, the main `video` could represent a live TV streaming video, and the area `a1` could be dynamically inserted or updated on-the-fly by the broadcaster to indicate an advertisement.

[5]https://alanlivio.github.io/paper-gingaf-acm-mm26/



(a) Clock 2.00 (green_bg starts). **(b) Clock 4.00 (red_bg starts).**

Figure 1: `video.ncl` execution (by `gingaf` on Windows).

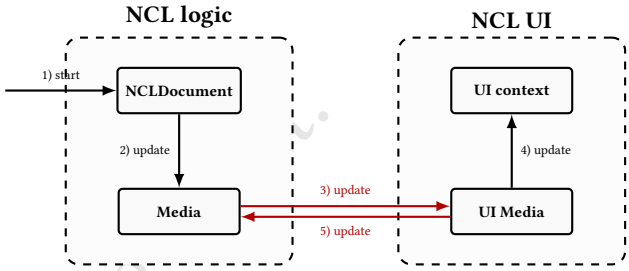


Figure 2: Related work communication diagram

3 Related Work

The development of NCL players has historically been led by the TeleMídia Lab at PUC-Rio[6], producing several versions. We discuss the two main ones below.

PUC-Rio Ginga v0.13.6 [6][7] was designed for resource-constrained early-generation digital TV receivers. It used SDL[8] as the UI context for rendering graphics and managing window events, and FFmpeg[9] for audio/video decoding. Its execution model relied on multiple threads and event listeners to handle asynchronous media playback: dedicated threads monitored AV decoder callbacks and posted events into the NCL engine, creating concurrency that was difficult to test deterministically.

PUC-Rio Ginga v1.0 [3][10] modernized the renderer by adopting GTK[11] as the UI context and GStreamer[12] as the AV decoding pipeline. A key architectural advance was the move to a mono-threaded model inspired by synchronous execution: rather than spawning per-media threads, the engine advances state through a central tick loop, reducing race conditions and improving predictability.

The above related work shares a common problem: both implementations focus on desktop environments and never had web support because of their strong library decoding dependencies. They are heavy, complex to build, and require platform-specific drivers for multimedia rendering. Figure 2 provides a simplified communication diagram that abstracts away libraries, threads, and callbacks to make clear that the NCL logic and NCL UI were tightly coupled. In it, `Media` objects directly reference `UI Media` components (arrow

[6]http://www.telemidia.puc-rio.br/
[7]https://github.com/TeleMidia/ginga/releases/tag/v0.13.6
[8]https://www.libsdl.org
[9]https://ffmpeg.org
[10]https://github.com/TeleMidia/ginga/releases/tag/v1.0
[11]https://www.gtk.org
[12]https://gstreamer.freedesktop.org

Unpublished working draft. Not for distribution.

3) and UI Media calls back into Media to report playback events (arrow 5). The red arrows highlight the fundamental problem: this bidirectional coupling makes it impossible to instantiate or test the NCL execution engine without an active rendering environment, preventing headless execution and deterministic debugging.

In contrast, gingaf introduces the NCL logic component as a fully self-contained package with no dependency on the NCL UI component. The execution engine exposes a clean tick() API, allowing it to run in-memory without any graphical context, enabling high-speed unit testing and step-by-step debugging of NCL synchronization logic.

Figure 3 illustrates how gingaf resolves this coupling. Unlike the related work, NCLDocument has no knowledge of any UI component and never holds a reference to NCLMediaState or any rendering object. The communication is strictly unidirectional: only the NCL UI sends messages to NCLDocument (arrows 2 and 5, highlighted in green). The NCL logic simply updates its internal Media state, and the NCL UI reads it back on demand. This asymmetry is the key difference from Figure 2, where the coupling runs in both directions across the boundary.

4 gingaf

gingaf is built using Dart and the Flutter[13] framework. Dart is a multiplatform-oriented language, while Flutter is a runtime that allows Dart applications to run on desktop, mobile, and web. At its core, Flutter is based on widgets with states[14]. gingaf is built around two main components (Figure 3): the NCL Logic, implemented entirely in Dart, acts as a self-contained execution engine that manages the NCL event state machine with no knowledge of any display context; and the NCL UI, implemented as a Flutter widget, drives the engine and maps its output to native platform widgets. We discuss the two components in the next sections.

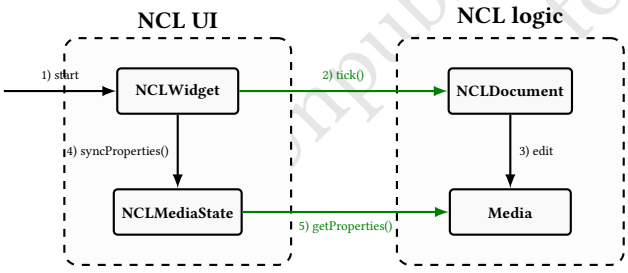


Figure 3: gingaf communication diagram

4.1 gingaf NCL Logic

The NCLDocument class is the core of the NCL logic component. The execution state of NCLDocument is managed through three key API methods: start() initializes execution; tick() advances the virtual clock and processes NCL Logic; and stop() halts execution.

The start() function begins by parsing the NCL XML input and validates the resulting tree against XSD rules, checking ID uniqueness and IDREF integrity. When start() ends, NCLDocument has the following data structures:

- DOM (Document Object Model) contains Element instances in the document structure (Figure 4). Each Element contains respective XML strings and its raw XML attributes. Node extends Element to hold runtime state (sleeping, occurring, or paused) and a property list. NCLua is a special Media that executes Lua scripts in pure Dart[15] for multiplatform (other elements were omitted for simplicity).
- VirtualClock: the engine maintains an internal logical clock which Media use for their elapsed time. This clock is advanced via tick() calls (ideal for debugging).
- ActionQueue: inside one tick(), actions are queued and drained deterministically to ensure consistent execution behavior.
- UIQueue: A shared queue that serves UI and logic without coupling them. At the end of a tick, the Document writes entries indicating which Media nodes changed visual properties, so the UI component can update rendering accordingly. Moreover, before each tick, the NCL UI component can write entries to notify the document that a media presentation has ended (e.g., a video file reached its end), allowing the engine to process the corresponding event and propagate any triggered links.

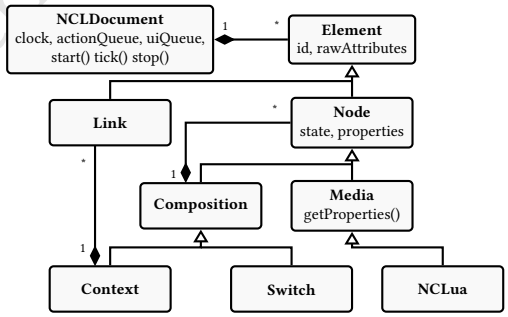


Figure 4: gingaf NCL DOM

tick() function advances an internal virtual clock by the given millisecond increment rather than relying on system time. This clock can be driven manually for testing or automatically by a Flutter periodic timer in a presentation environment. Decoupling time from the renderer enables high-speed headless testing: synchronization scenarios can be validated in-memory using inline XML strings, with no overhead from rendering windows, graphics libraries, or AV decoders.

On each call, tick() drains two queues. The ActionQueue collects transitions and link executions generated during state changes; these are drained deterministically in XML document definition order to ensure consistent execution behavior. The UIQueue serves communication across the component boundary: before the tick,

[13] https://docs.flutter.dev/
[14] https://api.flutter.dev/flutter/widgets/State-class.html
[15] https://pub.dev/packages/lua_dardo_plus

unpublished working draft. for distribution.

the NCL UI component writes entries reporting that a media presentation has ended, which the engine processes to propagate any triggered links; after the tick, the engine writes entries indicating which Media nodes changed visual properties, so the NCL UI component can update rendering accordingly.

Decoupling the execution from the renderer enables high-speed, headless testing. Synchronization scenarios can be validated in memory using XML strings, completely removing the overhead of rendering windows, loading graphics libraries, or initializing native audio/video decoders. An example of the execution log produced during headless execution is shown in Listing 2.

```
> dart run ncldoc/lib/main.dart ginga/examples/video.ncl
[ncldoc_cli] Executing ginga/examples/video.ncl in headless mode
[Clock: 0.0s] NCLDocument will start
[Clock: 0.0s] Node "body": sleeping->occurring
[Clock: 0.0s] Node "video": sleeping->occurring
[ncldoc_cli] perform 1 ticks per second indefinitely
[Clock: 0.0s] tickIndefinitely (1 ticks per/s)
[Clock: 1.0s] Tick
[Clock: 2.0s] Node "video" (area a1): sleeping->occurring
[Clock: 2.0s] Node "green_bg": sleeping->occurring
[Clock: 2.0s] Tick
[Clock: 3.0s] Tick
[Clock: 4.0s] Node "video" (area a1): occurring -> sleeping
[Clock: 4.0s] Node "green_bg": occurring -> sleeping
[Clock: 4.0s] Node "red_bg": sleeping->occurring
```

Listing 2: Log from headless execution of `video.ncl`

4.2 `gingaf` NCL UI

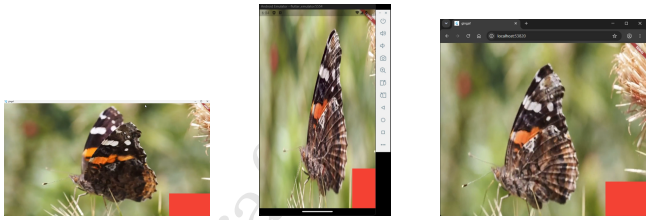
The `gingaf` NCL UI component is responsible for visual presentation. `NCLWidget` drives the engine by calling `tick()` on `NCLDocument` (arrow 2 in Figure 3) and delegates rendering to `NCLMediaState`, which reads the current visual properties of each `Media` node directly from the NCL logic component (arrow 5). The interface is composed of a root `Ginga` widget that utilizes a Flutter `Stack` to layout widgets. A background audio/video widget (`main_av_widget`) is placed at the bottom, and the active media elements are rendered on top within the NCL presentation surface. The `WidgetFactory` maps NCL elements to native Flutter widgets.

Currently implemented widgets include: `ImageWidget`: Displays images and handles image format caching; `AVWidget`: Renders native video/audio playback, supporting state tracking; and `TextWidget`: Renders formatted text blocks using bounds properties. Each widget extends `BaseWidget`, which maps NCL layout properties (like `left`, `top`, `width`, `height`, and `background`) to Flutter layout parameters (`Positioned` and `BoxDecoration`).

A key design aspect of `gingaf` is that `NCLWidget` holds an instance of `NCLDocument` and registers a Flutter periodic timer that periodically calls `doc.tick()` to advance the virtual clock. When a media widget experiences a state change (for example, when an `AVWidget` reaches the end of its video file), the widget notifies the `NCLWidget` via the UI Queue. The application then retrieves the corresponding `Media` node from the document and triggers its transition event, allowing the headless execution engine to run link conditions and propagate state changes back to the widgets. This ensures a clean separation of concerns: widgets only worry about rendering and local lifecycle events, while the engine handles the declarative NCL coordination logic.

5 Results

To show the correctness and multiplatform execution capabilities of `gingaf`, we execute the sample NCL application (`video.ncl`) described in Section 2. As a result, Figure 5 presents the execution of this application across three different environments: Windows, Android, and Web. Additionally, the temporal execution screenshots are shown in Figure 1 and the corresponding logs are provided in Listing 2.



(a) Windows. **(b) Android.** **(c) Web.**

Figure 5: `video.ncl` running on Windows, Android, and Web.

The reliability of the codebase is maintained through an automated test suite. The `ncl_doc` logic package contains 64 unit tests validating the parser, XML parsing rules, virtual clock progression, and deterministic link execution. The `gingaf` presentation package contains 14 tests verifying widget rendering, layout constraint resolution, focus state transitions, and integration with the NCL execution engine. All tests run in milliseconds due to the headless execution capabilities of the engine.

To replicate our results, we guide the reader to execute the commands below:

```
$ git clone -b v0.1.0 https://github.com/ginga-org-br/gingaf
$ cd gingaf/ncldoc
$ make deps
# Headless execution of video.ncl
$ dart lib/main.dart ../ginga/examples/video.ncl
# UI execution of video.ncl
$ cd ../ginga
$ make deps # See SETUP.md for environment configuration
$ flutter run -d <chrome,windows,emulator-XXXX> --dart-define="APP
    =examples/video.ncl"
```

6 Final Remarks

`gingaf` demonstrates that decoupling the NCL logic component from the NCL UI component provides a flexible, testable, and highly portable solution for executing NCL digital television applications. The NCL logic component, implemented as `ncl_doc`, directly addresses **P2 (Lack of NCL Debugging Features)**: it runs fully in-memory with no rendering dependency, enabling deterministic unit testing of NCL synchronization logic. The NCL UI component, built on Flutter, directly addresses **P1 (Lack of Accessible NCL Development Environment)**: it compiles to Windows, mobile, and web from a single codebase, removing the need for specialized hardware or platform-specific setups.

Next steps include the development of a web-based NCL playground that integrates debugging features to allow developers to run and step through NCL applications directly in the browser without any local installation, closing the accessibility gap for NCL developers and researchers.

Unpublished working draft. Not for distribution.

References

[1] ABNT. 2023. *ABNT NBR 15606-2: Digital terrestrial television – Ginga-NCL for fixed and mobile receivers – XML application language for application coding.* Technical Report. Associação Brasileira de Normas Técnicas.

[2] Cosette Castro and Cristiana Freitas. 2016. Brazil 4D: An Experience of Interactive Content Production for Free-to-Air Digital Television. In *Handbook of Research on Comparative Approaches to the Digital Age Revolution in Europe and the Americas.* IGI Global Scientific Publishing, 202–215.

[3] J. P. Dodsworth, L. d. M. Terças, A. L. V. Guedes, G. F. Lima, C. d. S. S. Neto, and S. Colcher. 2018. Dynamic Integration of Foreign-Language Parsers into an NCL Player. In *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web.* doi:10.1145/3243082.3243095

[4] Joel dos Santos, Li-Chang Shuen Cristina Silva Sousa, Carlos Soares Neto, Guido Lemos, Sérgio Colcher, Marcelo F. Moreno, and Débora Muchaluat-Saade. 2026. An Overview of the TV 3.0 Application Coding Layer Specification. *IEEE Transactions on Broadcasting* PP, 99 (2026), 1–15. doi:10.1109/TBC.2026.3689351

[5] ITU. 2009. *H.761: Nested Context Language (NCL) and Ginga-NCL for IPTV Services.* Technical Report. ITU, Geneva, Switzerland.

[6] Marcio Ferreira Moreno. 2010. *Conciliando Flexibilidade e Eficiência no Desenvolvimento do Ambiente Declarativo Ginga-NCL.* Ph. D. Dissertation. Pontifícia Universidade Católica do Rio de Janeiro. doi:10.17771/PUCRio.acad.35200

[7] Luiz Fernando Gomes Soares, Rogério Ferreira Rodrigues, and Romualdo Monteiro de Resende Costa. 2006. Nested Context Model 3.0 Part 6 – NCL (Nested Context Language) Main Profile. (2006).

Unpublished working draft.
Not for distribution.