State-based Layering

A conceptual framework for live production of time-driven media experiences on data-driven consumer platforms

Ingar M. Arntzen[1], Njål T. Borch[2], and Anders Andersen[3]

[1] NORCE Norwegian Research Centre, Norway,
inar@norceresearch.no
[2] Schibsted, Norway,
njal.borch@schibsted.com
[3] UiT The Arctic University of Norway,
anders.andersen@uit.no

Abstract. Web-based technologies are widely used as production tools for time-driven media and as interfaces for live reporting of time-sensitive developments. Time-driven media face growing demands for interactivity, adaptation, and personalization, suggesting a more web-like model where layer orchestration and time-sensitive rendering increasingly occur on the client side, within data-driven consumer interfaces. Yet realizing this shift is challenging. Time-driven media production relies on precise control over layering—when layers are activated, deactivated, and how they are combined and displayed. In data-driven platforms, however, support for external, time-sensitive control is weak, and layer orchestration and playback are handled by distinct media frameworks—each constrained to a particular application context and technology stack. Our approach is to shift the problem from media frameworks to the data model. By supporting production-controlled layering and playback within application state, time-driven media experiences can be realized as native expressions of data-driven platforms, fully exploiting available rendering technologies and backend services. The main contribution is State-based Layering (SbL), a conceptual framework for developing controllable, time-dependent application state. SbL generalizes concepts defined within existing media frameworks, promoting them as generic programming constructs—independent of data formats and user interfaces, and applicable across diverse application contexts. StateLayers, a reference implementation of SbL, shows that timing, control, and layering can be fully encapsulated within application state and remain compatible with modern development workflows. SbL marks an important step toward scalable personalization, enabling production systems to directly shape unique time-driven narratives for individual users through real-time control over consumer interfaces.

Keywords: time-driven media, data-driven media, production control, layered composition, state-based rendering, web, personalization

List of abbreviations

UI User Interface
API Application Programming Interface
HTML HyperText Markup Language
DOM Document Object Model
CSS Cascading Style Sheets

1 Introduction

Time-driven media productions (e.g., audio and video) face growing demands for advanced features, including interactivity, multi-device support, and individual adaptation (e.g., personalization, customization, accessibility). Meeting these demands at scale requires that certain aspects of production be shifted into consumer interfaces, where it can be realized with sensitivity to local context and user preferences, and leverage data-driven technology platforms and capabilities of specific device types. For example, Object-based Media distributes media as objects, enabling flexible assembly and rendering in consumer-side web interfaces [8, 15].

We share this ambition but follow the approach of *Control-driven Media (CdM)* [12], which describes a more web-like model for time-driven media production, characterized by (i) client-side assembly of multiple dynamic data and control sources, and (ii) active production control through those sources, enabling production systems to actively shape lean-back, time-driven narratives through precisely timed interface updates. CdM promotes consumer interfaces as active components of the production infrastructure, and preserves the traditional semantics of studio-based production, where producers can actively control every aspect of the user experience in real time. Flexibility also increases, as lean-back, personalized narratives can be directed in real time from unique combinations of data sources and media contents, perhaps assisted by online AI-based production agents.

However, a remaining challenge in CdM concerns the assembly of time-dependent data and control resources into coherent user experiences. We refer to this as *temporal layering*, or simply *layering*. Production must be able to time-shift and combine data layers, switch between layouts and content sources, and schedule delayed transitions. Control state—such as *zoom level*, *viewport position*, *audio level*, or *playback offset*—must be adjustable, and rendering must remain consistent with timeline progression and responsive to changes in data and control state. Moreover, solutions should be practical and integrate seamlessly within data-driven platforms.

This, though, is not easily achieved. Support for temporal layering and playback is primarily available through embedded media players and special-purpose media frameworks. Such reliance is problematic, as these frameworks are typically confined to a single technology stack, application context, and usage pattern, and are difficult to combine or extend. Similar limitations apply to production control, which may be implemented through feeds, streams, or real-time

data sharing, but still depends on integration with player environments for time alignment.

To address this, we propose shifting the responsibility for layering away from UI components and media frameworks and into the data model. In this approach, the effects of layered media production materialize in application state, and can be observed and rendered across diverse UI components and media frameworks. Moreover, a generic solution for layering in application state could encapsulate complexities, enable integration with application-specific data models, and be applicable across a wide range of applications.

We introduce *State-based Layering (SbL)*, a conceptual framework for layering and playback in application state, as a foundation for time-driven media production on data-driven platforms. SbL defines programming concepts *Track* and *Cursor* as unifying abstractions for resources with *time-dependent* or *current* state, and provides a set of layering operations, including *record*, *playback*, *merge*, *shift*, and *transform*. SbL enables application-specific production logic to be expressed within a state-based programming model, addressing challenges related to timing, online production control, and layered composition.

While layering is a well-established concept in time-driven media production, this work addresses it in a novel context, as an aspect of application state management in data-driven platforms, decoupled from media formats, processing environments, distribution protocols, and rendering technologies. Programming constructs defined in SbL are generalizations over similar concepts found in existing media frameworks.

This paper focuses on making a clear conceptual contribution, laying the foundation for future empirical research. Its scope is therefore limited to concept design and functional validation. Broader implications for systems architecture and application design have been addressed in prior works [12].

The paper is organized as follows: Section 2 examines current support for time-driven media production and key challenges. Section 3 defines the approach and the research problem. Section 4 presents SbL as a conceptual framework, with key concepts detailed in Section 5. Section 6 presents the evaluation, and Section 7 provides additional insights, before the paper is concluded in Section 8.

2 Background

Our objective is to support centrally controlled production of time-driven media experiences natively within interactive, data-driven, consumer interfaces. This section examines the nature of layered media production and identifies key challenges relative to this objective.

2.1 Layering in Time-driven Media

Temporal layering is a central concept in time-driven media production, supporting the combination of multiple time-dependent resources into a single, serialized

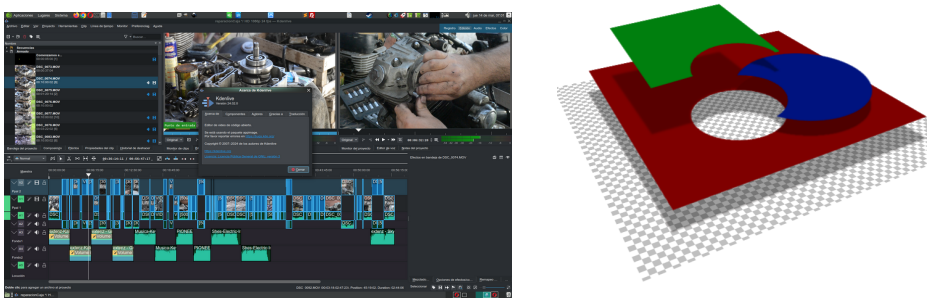

Fig. 1: *(left)* Track layering in video editor: multiple video clips orchestrated along a timeline. *(right)* Image layers in a photo editor. Images by Equipo de desarrollo de Kdenlive and David.atwell http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons.

media experience (see Fig. 1 *left*). In music production, for example, audio mixers combine multiple audio channels into a unified output, while Digital Audio Workstations (DAWs) enable the scheduling of samples and tracks, with support for transitions such as crossfades [6, 14]. These workflows are often iterative, with new tracks recorded in alignment with existing ones. They can also be recursive, in the sense that an input layer may be the output from an earlier layering process. Video production systems similarly rely on layering, integrating video, audio, and graphical assets through tools such as video editors [4, 5] and frameworks for motion graphics and animation [1, 16]. Live production systems support dynamic orchestration of layers and layouts: switching between live and pre-recorded content sources, activating and deactivating graphics layers (e.g., lower thirds or on-screen logos), and selecting layouts such as side-by-side or overlay. Layering is also evident in distribution. Media container formats such as MP4, MKV, and WebM include multiple synchronized streams (video, audio, subtitles, graphics, metadata), preserving the layered structure for playback systems. Object-based Media [8, 19] offers even more flexibility by representing media as a stream of objects, including media content, structure, and meta-data.

2.2 Layering in Data-driven Media

Layering is also a common abstraction in data-driven systems, where objects and data sources are loaded into the client and assembled dynamically in the interface (see Fig. 1 *right*). For example, image editors like Photoshop and GIMP [3, 21] allow users to build composite images from stacked, partially transparent layers, and to edit, reorder or hide layers during the authoring process. In map visualization frameworks such as Google Maps and OpenLayers [22, 26], vector overlays and satellite imagery are organized as separate, composable lay-

ers within a shared geographical model. Many UI frameworks support layering in layout: including the DOM model in the web platform [32], Android's View/ViewGroup [24], and iOS's UIKit [7]. In web development, layering effects can be controlled by CSS properties like z-index and opacity. These parameters can also be modified at runtime, enabling animated layout-transitions. Rich media frameworks such as Adobe Animate [2] support timeline-based composition of graphics layers. In HTML5, synchronized video overlays can be supported through TimedTextTracks [33]. The Web Audio API [29] supports low-level audio programming in the browser, allowing scheduling of audio samples and configuration of filters and effects within an audio processing graph.

2.3 Layering as A Common Pattern

Despite differences between time-driven and data-driven approaches, layering mechanisms share important characteristics. In both models, layering provides a powerful and intuitive metaphor for media production, yielding predictable outcomes while encapsulating technical detail. The layering metaphor remains relevant across different stages of the production pipeline and at varying levels of abstraction, supporting a compositional production model. Layering captures the essence of media production: a transition from high narrative potential, rooted in available resources, to a specific realization as those resources come together in a single, coherent user experience. As such, we regard layering as a common pattern across time-driven and data-driven technologies.

2.4 Challenges

To facilitate our objective of centrally controlled media production within consumer interfaces, we identify two distinct challenges.

Control. The first challenge concerns how a production system can exercise precise, detailed control over consumer-side media experiences. Control commands or events can be pushed through feeds, streams, or services for real-time data sharing, yet these solutions are often limited to live control scenarios and application-specific command formats. EaseLive [18], for instance, provides interactive graphics on client devices for live TV productions, using a cloud-based solution for distribution of control streams. Game engines such as Unreal and Unity [27, 28] provide mechanisms for highly dynamic, real-time game control, but solutions are tightly bound to platform and game logic. TimingObject and SharedMotion [13, 30] provide real-time, distributed time-control and synchronization, but do not address production control beyond clock sharing. StateTrajectory [11] has been proposed as a more general solution, supporting application-defined control state in both real-time and time-shifted scenarios.

Integration. The second challenge concerns how layering can be realized within data-driven platforms without imposing unwanted restrictions. Currently, support for temporal layering and playback is typically provided by embedded media players, media frameworks, or standalone applications for audio and video

editing. Such frameworks are usually bound to specific domains (music, video, animation), usage scenarios (live or on-demand), and fixed technology stacks (data format, distribution method, user interface). External production control can also be difficult to integrate, as client-side media frameworks are often designed for interactive control by users. Additionally, media frameworks can be difficult to combine, update, or extend, as they are often constrained by formal standards and dependent on third-party maintenance. These limitations prevent time-driven media from fully exploiting the capabilities of data-driven platforms.

2.5 Vision

In contrast, we envision a shift where control, layering, and playback are no longer regarded as framework concerns, but addressed in application state and supported by generic programming constructs. Such an approach could elevate layering from a framework-specific feature to an application-level concept, with effects spanning UI components, devices, and platforms. It could also establish a common methodology for developing and controlling time-driven media experiences natively on data-driven platforms.

3 Approach

To integrate layering within data-driven platforms, we adopt a reactive, state-based approach (see Fig. 2). This implies a clear separation between data model and UI, where the UI always reflects the current state of the data model and re-renders automatically when the data model changes. By treating the data model as a single source of truth, consistency can be ensured across all UI components. We further adopt a design pattern where application-specific layering is handled within the data model, not in UI components.

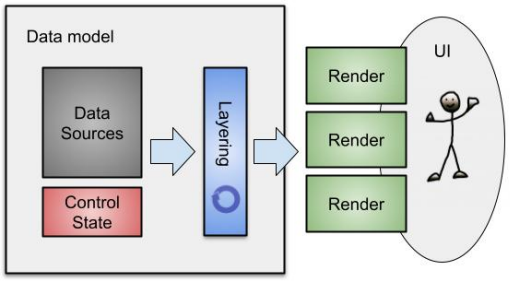

Fig. 2: State-based approach with layering. Data sources *(black)* and control state *(red)* form the data model. The layering component *(blue)* transforms data and control state into layered output state in preparation for rendering. Changes in the data model propagate through layering to render components *(green)*.

As a simple example, consider multiple data tracks competing for screen time. Layering could be used to merge these into a single output track, based on an application-specific policy. This simplifies the render component, which only needs to observe the output track, and can remain the same even if details of the merge operation are modified. Similarly, given a subtitle track, layering functionality could expose the active subtitle based on timeline progression. This way, UI components can remain simple and data-driven, yet still render the correct subtitle at any time.

Further implications of a state-based approach are outlined in Appendix A.

3.1 Problem Statement

Based on challenges identified in Section 2 and the approach outlined above, we have defined the following objective: Define a conceptual framework for temporal layering in state-based applications. The framework shall:

P1 Enable layering logic to be defined within the data model
P2 Support consistent representation of time-dependent applications state
P3 Be generic, flexible, expressive, and extensible
P4 Be practical, intuitive, and broadly applicable
P5 Support dependable and efficient implementation
P6 Support integration with online services for real-time state sharing
P7 Support usage with different rendering frameworks

4 State-based Layering

We define *State-based Layering (SbL)* as a directed processing graph. Developers create layering functionality by combining and transforming application resources into new graph nodes, using predefined layering operations such as *merge*, *transform*, *record*, and *playback*. The graph is dynamic and extensible, allowing for runtime changes to layering logic.

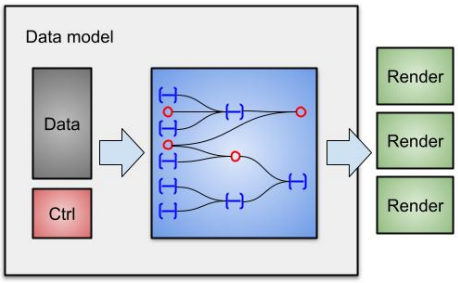

Fig. 3: Layering graph with tracks *(blue brackets)* and cursors *(red circles)*.

Fig. 3 illustrates this layering graph. Input nodes (left) represent application resources: data sources (black) or control state (red). Resources with a *time-dependent* state are represented as *tracks (blue brackets)*, whereas resources with a *current value* are represented as *cursors (red circles)*. *Layering operations* define state transitions within the graph, enabling the creation of new tracks and cursors from existing nodes. Rendering components (green) can observe any node in the graph. State changes propagate through the graph from left to right.

Before describing key concepts of this layering graph in Section 5, we first illustrate its use through examples.

Content Selection and Scheduling

Selection and orchestration of content sources is a key objective in consumer-side media production. Consider a scenario with one main video stream (e.g., the official broadcast) and several auxiliary streams (e.g., alternate camera angles). A production system may aim to display at most two streams at a time, dynamically assigning them to two fixed display elements.

SbL can support this kind of production-controlled experience. For example, production could maintain two official tracks representing stream selections, one for each display element. As production selects a new source URL for a display, the URL is appended to the corresponding track. Coverage could also be personalized by overlaying user-specific tracks on top of official ones. This would allow certain camera angles to be prioritized based on team affiliation, or promoted to the primary display element during key moments. Additional tracks may reflect stream selections made by the user or by collaborators in a shared session. Layering logic would then serve as a mechanism for reconciling alternative tracks of stream selection in accordance with production goals. Importantly, the layering graph would encapsulate this complexity, effectively providing a ready-made, personalized production stream for each display element.

Conflicts

Conflict resolution is another important aspect of layered media production. For instance, there might be conflicts between subtitles, lower-third graphics, and superimposed elements, as render components may compete for shared or limited screen time or display area. Increased levels of personalization and variation may also generate new conflicts, requiring resolutions to be sensitive to a unique context.

SbL can be used to avoid conflicts and implement resolutions. For example, a new track can be computed as the logical *AND* of two conflicting text tracks, marking time regions where both are active. This logical track can then drive specific adaptations in layout, such as resizing components or repositioning them to avoid overlap. Alternatively, conflicts between tracks may be resolved

by *merging* them into a single track, for instance stacking tracks by priority or aggregating overlapping inputs. By addressing conflict resolution as part of the layering graph, as opposed to case-by-case adaptations in UI components, solutions can provide stronger guarantees and remain effective across different resource types and combinations.

Metadata

Metadata serves many purposes in media production, including enhanced navigation, search, and interactivity. In audio and video contexts, metadata can describe the structure of chapters and scenes, camera focus points, volume levels for different audio tracks, or the on-screen positioning of objects and characters.

SbL enables more flexible and practical use of metadata by representing them as tracks within the application state. For example, a video player might offer a more intimate viewing experience by automatically zooming in on the person speaking. This can be achieved dynamically by adjusting zoom level and viewport offset during media playback, based on a viewport-control track derived from metadata such as face detection or audio intensity. Different styles of viewport control can also be supported tailored to specific content types or user preferences. Crucially, layering makes it easier to incorporate out-of-band metadata, and supports the development of more sophisticated behaviors through further combination and customization.

Playback

Presenting data in accordance with timeline progression and media control is a key challenge in consumer-side media production. For example, there might be multiple data and meta-data tracks which need to be rendered in synchrony, across a handful of rendering components. Some tracks may need to be time-shifted, to match the local timeline or other tracks.

SbL models time-shifting and playback through designated layering operations. For example, misalignments between tracks can be addressed in the layering graph by applying *shift* operations to particular tracks. Synchronized playback is ensured by applying the *playback* operation to multiple tracks, using a shared time control. An additional playback cursor (i.e., *pre-cursor*) can also be set up to serve upcoming stream-url's 1-2 seconds ahead of time. This would allow loading and initialization to be completed ahead of time, supporting seamless transitions between streams. The pre-cursor would simply be a different playback cursor, running on a time-shifted media clock.

5 Key Concepts

This section introduces key SbL concepts, including resource abstractions, *Track* and *Cursor*, and various layering operations needed for a flexible and expressive layering graph.

5.1 Track

Track is an abstraction over application resources whose state is organized along a timeline. Common examples include time-referenced datasets (e.g., logs, subtitles, cues, GPS tracks, feeds) or control state (e.g., playlists, command sequences, music scores, waypoints).

> ***Definition:*** *A Track represents an application resource with state defined in reference to a timeline. For any given timeline offset, a Track has either no value (undefined) or one value. Tracks can be typed, meaning that all values have the same type (e.g., int, float, string, object, tuple, list, etc.). Tracks can be empty, meaning that no value is defined for any timeline offset.*

Fig. 4 shows three example tracks, each depicted as a pair of blue brackets enclosing a set of values arranged horizontally along a timeline. Track 1 defines CSS color codes (red, blue, green, black). Track 2 shows audio volume (float) undergoing both discrete and gradual adjustments over time. Track 3 captures snapshots of a list of characters, where some characters have been added or removed at different points on the timeline.

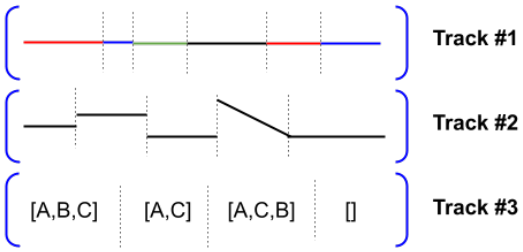

Fig. 4: Track examples. *#1* CSS color codes, *#2* volume level, *#3* list of characters.

The *Track API* defines a minimal set of methods for accessing track values and observing state changes. It serves as a read-only interface to underlying resources. Requests for state change must be directed to underlying resources.

Track API:
- `track.query(offset)`: Returns the value corresponding to the given timeline offset, or undefined if no value is defined.
- `track.on('change', callback)`: Registers a callback for state changes in the track.

5.2 Cursor

Cursor is an abstraction over application resources with a current value. Common examples include observable variables, object properties, live sensor readings, clocks, animated transitions, interpolated states, or playback state.

Definition: *A cursor represents an application resource with a current value. The value may be undefined. Cursors can be typed, meaning that cursor values are restricted to a single type (e.g., int, float, string, object, tuple, list, collection). Cursors used for time control are restricted to number types. Cursor values may change either discretely or dynamically.*

Cursors distinguish between two modes of state change: discrete and dynamic. In discrete mode, cursors behave like classical programming variables, where the value remains static until reassigned. State changes can be observed by listening for change events. In dynamic mode, cursors represent continuous state change, such as animations or dynamic sensor readings. In this mode, state changes can be observed through repeated polling. By supporting both modes, cursors can accommodate a wide range of application concepts and also support transitions between discrete and dynamic behaviors. The term cursor is intended as a metaphor for state change over time as the cursor glides along a track.

The *Cursor API* defines a minimal set of methods for accessing the current value and observing state changes. It serves as a read-only interface to underlying resources. Requests for state change must be directed to underlying resources.

Cursor API:
- `cursor.value()`: Returns the current value of the cursor, or undefined.
- `cursor.dynamic()`: Returns true if the cursor is currently undergoing dynamic value change, otherwise false.
- `cursor.on('change', callback)`: Registers a callback for cursor state changes.

5.3 Layering Operations

Layering operations implement state transitions within the layering graph (see Fig. 3). For example, a 1:1 operation *map* implements a custom value transition on a track, whereas *shift* repositions track values on the timeline by a fixed offset. N:1 operations *merge* and *select* create a single track from a set of tracks.

Logical operations *AND, OR, XOR* support track expressions. For example, two tracks combined with an AND operation produce an output track that is defined only in regions where both input tracks are defined. N:M operations can capture more complex dependencies between input and output tracks.

> ***Definition:*** *Layering operations implement state transitions, from a set of input nodes to a set of output nodes. Input and output nodes can be both track and cursor types. State changes in input nodes trigger reevaluation of the layering operation, leading to state changes in output nodes.*

5.4 Merge

Combining multiple input tracks into a single output track introduces a particular challenge. Input tracks may overlap on the timeline, yet the output track is restricted to a single value (or undefined) for any timeline offset. Fig. 5 illustrates this, showing three input tracks (red, green, blue) (bottom) combined into a single output track (top). Regions with contributions from more than one input track have multiple colors (red/blue, green/blue).

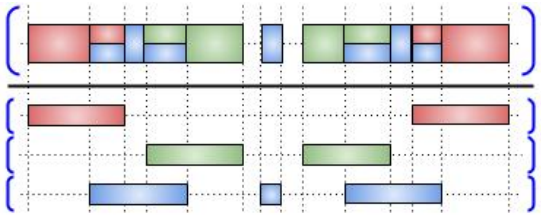

Fig. 5: Merging three input tracks (bottom) into one output track (top).

The *merge* operation resolves this by accepting a custom value function that specifies how track values are calculated from multiple inputs. For example, value functions may add all contributions or simply select the top-most contribution, thereby implementing layering operations *sum* and *stack*.

5.5 Playback and Recording

Track playback is modeled as a state transition. Fig. 6 illustrates playback, transforming an input track of colors (bottom) into an output cursor with a color value (top). The output cursor will always assume the correct color from the input track, in accordance with timeline progression and interactive time control. Currently, the value of the output cursor is green, but it will change in a few moments as the timeline offset increases.

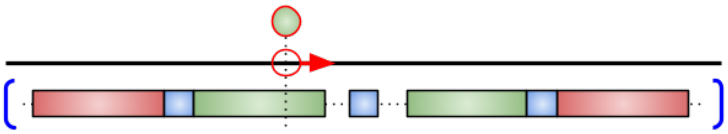

Fig. 6: Playback (red circle and arrow) along the input track (bottom) produces an output cursor (top).

Similarly, track recording can be defined as a state-transition. Given a live input cursor (e.g., color) observed changes can be time-stamped and appended to an output track (e.g., colors).

Fig. 7 illustrates playback and record as layering operations. Note that the timeline offset (i.e., playback offset or recording clock) can also be modeled as a cursor. This implies that playback is an operation with two inputs (track and cursor) and one output (cursor), whereas recording is an operation with two inputs (cursors) and one output (track). Crucially, this highlights playback and record as generic operations, reusable across different types of tracks and cursors in the layering graph. Playback and record are also inverse operations, bridging the gap between representations for time-dependent state (track) and current state (cursor).

target = playback(source, control)
- **source:** Playback source (track)
- **control:** Time control (cursor)
- **target:** Active source value (cursor)

target = record(source, control)
- **source:** Recording source (cursor)
- **control:** Time control (cursor)
- **target:** Recorded values (track)

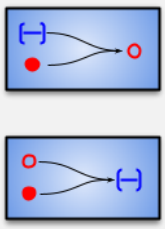

Fig. 7: Playback (top) and record (bottom) as layering operations.

6 Evaluation

This section evaluates State-based Layering (SbL), focusing on its quality as a conceptual framework. The evaluation draws on a concrete implementation of SbL (Section 6.1) and a proof-of-concept demonstration (Section 6.2). The results are presented in Section 6.3, organized across five complementary levels, referencing objectives from the problem statement (Section 3.1).

- `Architecture`: how SbL integrates with data-driven platforms (P1, P6, P7).
- `Methodology`: how SbL provides a flexible programming model for time-sensitive media applications (P3, P4).
- `Concepts`: how SbL concepts capture key ideas, encapsulate complexity, and provide explanatory power (P3, P4).
- `Mechanism`: how SbL ensures consistency in layered output state (P2, P6).
- `Framework`: how SbL can be implemented as a practical and dependable software framework (P5, P6, P7).

A detailed evaluation of performance and scalability is left to future work exploring SbL in real-world applications.

6.1 Implementation

StateLayers, a reference implementation for SbL, has been developed and released on GitHub [9]. The framework is implemented in *JavaScript*, enabling the development of layered application state on the Web platform and in *Nodejs* environments. The framework is lightweight and portable.

StateLayers supports integration with external resources, through implementation of the StateProvider interface, which is simple and generic in nature. StateProviders act as proxies for local or online-hosted resources. This allows track and cursor state to be shared in real-time across interfaces, devices or platforms.

StateLayers includes a set of pre-defined layering operations, such as *stack*, *sum*, *merge*, *shift*, *record*, and *playback*, along with logical operations *AND*, *OR*, *XOR*, and *NOT*. StateLayers is also extensible, allowing custom layering operations to be created when needed. Layering graphs are formed implicitly through layering operations, as new, derived tracks and cursors are generated from existing ones. Nodes in the graph are loosely coupled, opening up for flexible reconfiguration. For instance, a playback cursor may be assigned a new track source or a new time control, by resetting its *src* and *ctrl* properties. Synchronized track playback is implied if playback operations use the same time control.

StateLayers employs indexing techniques and caching to ensure efficiency. Caching is particularly beneficial with respect to playback and sampling of tracks, as these operations imply a high degree of temporal locality. To avoid state duplications, derived tracks are often implemented as stateless views.

6.2 Demonstration

A simple, 100 second, animated web experience is crafted as a proof-of-concept demonstration. The presentation emulates stylistic conventions of a video-based experience, including fixed aspect ratio, scene changes, shifts in content and layout, activation of superimposed graphics and subtitles, and a progress bar with playback controls.

The main objective is to demonstrate how SbL can provide new opportunities for flexibility and control in time-driven web experiences. The data model

```
const colors = sl.track({items: [
    {itv: [0, 1], data: "red"},
    {itv: [1, 2], data: "blue"},
    {itv: [2, 3], data: "green"},
    {itv: [3, 4], data: "yellow"},
    ...
]});
const ctrl = sl.object({numeric:true});
const color = sl.playback({src:colors, ctrl});
color.on("change", (state) => {...})
ctrl.motion({velocity:1})
```

Fig. 8: Track playback in five statements: (i) define a track of colors (*itv* denotes time interval), (ii) define a time control, (iii) define a color cursor through playback of the colors track, (iv) render change events from the color cursor, and (v) start playback with the time control. The displayed color will change during playback, and respond to changes in time control or track state.

includes a variety of tracks pertaining to different aspects of the experience. For example, the page layout is set up to show 5 predefined floating panels *(main, second, super, title, dog)*. For each of these panels there is a track describing *position*, *size* and *z-level*. Similarly, there are tracks for content channels *(subtitles, images, super texts)* and rendering components *(text viewer, image viewer)*. Given the ability to map a content source and a render component to each panel, this effectively corresponds to a rudimentary production system. Moreover, by realizing these mappings as tracks, they effectively become time-dependent production scripts directing how content selections and configuration parameters will change during the experience.

Rendering involves synchronized playback of all tracks, driven by a common time control. Output cursors from playback operations are provided as inputs to state-based ReactJS [20] components, requiring only a simple React hook for integration. Furthermore, the ability to manage application state as a set of independent resources (tracks and cursors) represents a source of considerable flexibility. It means that key aspects of the production (capture, storage, distribution, playback, sharing scope) may be addressed on a per-resource basis, as opposed to per-experience. For example, control interfaces for panel placement or content selection could be developed as separate web pages, without requiring access to the rest of the data model. Moreover, the ability to render directly from a set of independent resources, without conforming to a common data model or format, is liberating.

6.3 Results

Architecture. At the architectural level, SbL demonstrates a solution to layering that is confined to the data model *(P1)* and supports integration with

custom backend services and rendering technologies *(P6, P7)*. SbL achieves this by introducing an independent software layer between the data model and UI, or technically, as an application-defined extension of the data model. This design enables integration with online services for real-time sharing of data and control state, and usage with modern state-based rendering frameworks. Our proof-of-concept demonstration confirmed this architectural flexibility by using an in-house solution for state sharing and React [20] for rendering. Crucially, neither the backend service nor the rendering framework required modification, yet could still support a time-driven user experience. Moreover, support for distributed production control was demonstrated through real-time editing of online control state.

Methodology. At the methodological level, SbL demonstrates a generic, flexible, and expressive approach to time-driven media production *(P3)*, consistent with established patterns for application development on data-driven platforms *(P4)*. Unlike traditional approaches, time-driven application behavior is not delegated to media frameworks, but expressed in application code and supported by appropriate programming constructs. SbL defines a programming model for time-dependent application state, based on uniform resource abstractions (Track and Cursor) and a set of generic layering operations (shift, merge, playback). Our demo application confirms the flexibility and expressiveness of this approach *(P3)* by realizing a rudimentary production system for time-driven media experiences as a Web app. Practicality *(P4)* is demonstrated through application code, characterized by simple statements defining resources and layering operations. Moreover, SbL's emphasis on shifting application-specific logic away from UI components and into the data model mirrors design goals of modern reactive programming frameworks such as React and Angular [20, 23].

Concepts. At the conceptual level, SbL provides intuitive and generic programming constructs that ensure broad utility *(P3, P4)*. The *Track* and *Cursor* abstractions capture the fundamental distinction between entities with *time-dependent state* and those with *current state*, and are agnostic to properties such as *data type*, *role* (data or control), *origin* (local or remote), and *change type* (discrete or dynamic), making them broadly applicable *(P4)*. Layering operations define an extensible suite of reusable state transitions *(P3)*. Expressiveness is achieved through composability and verified through unit tests and the demo application. These concepts also mirror abstractions found in other media frameworks, suggesting that they are familiar and intuitive, while still encapsulating important complexity. Finally, by virtue of their generic and minimalist design, these concepts provide a valuable foundation for reasoning about structural differences across media systems and applications.

Mechanism. At the mechanism level, SbL ensures consistent distribution of time-dependent data and control sources *(P6)* and consistent composition of layered application state *(P2)*. Track state is defined as a set of non-overlapping time functions, ensuring a well-defined state along the timeline. This enables reliable

sharing of time-dependent state across a network and across time-frames, enabling time-consistent playback at the client side *(P6)*. This forms the basis for time-sensitive production control in consumer interfaces. Temporal consistency is further preserved through layering operations, and state changes propagate automatically, ensuring that all tracks and cursors remain consistent with their underlying resources, including time progression *(P2)*. Consistency is verified through unit tests for tracks, cursors, and layering operations.

Framework. At the framework level, SbL can be implemented as a dependable and lightweight framework *(P5)* that integrates naturally within modern web development workflows *(P6,P7)*. This is demonstrated by StateLayers, a reference implementation of SbL in JavaScript. StateLayers enforces well-defined track state by preventing non-overlapping state definitions, and implements precise cursor playback using a high resolution local clock (or a proxy to an external clock) combined with a precise timeout mechanism *(P5)*. Integration with custom data sharing-services is demonstrated for an in-house service for real-time data sharing *(P6)*. Moreover, React, a widely used rendering framework, was integrated with cursors through a React hook *(P7)*. StateLayers constitutes a thin wrapper over select application resources and does not introduce significant overhead or limitations beyond those inherent to data-driven platforms [11].

7 Discussion

Reactive Programming. State-based Layering (SbL) aligns with the reactive programming model popularized by modern UI frameworks such as React, Vue, Svelte, and Angular [20, 23, 25, 34]. These frameworks are primarily designed for real-time rendering of dynamic data and interactive control state, while offering limited support for time-dependent orchestration and rendering. SbL does not compete with these frameworks, but instead extends the data-driven model with fundamental concepts for building time-dependent media applications, including live or time-shifted production control, and synchronized playback of application state. This opens a new application domain for reactive rendering frameworks.

Scalable Personalization. A key challenge in time-driven media production is reconciling personalization with scalability. Broadcast and streaming platforms offer exceptional scalability, but their production models are centered on distributing a single, finalized media asset to a broad audience. In contrast, data-driven platforms, such as Web interfaces and native apps, offer much stronger support for variation and differentiation across consumers. This model emphasizes the production of reusable parts, hosted by servers and assembled dynamically in consumer interfaces. Future demands in media production will likely require a combination of both models. SbL offers a promising bridge, by (i) enabling time-driven media production natively within data-driven interfaces, and (ii) reconciling centralized production control with decentralized assembly and rendering. Moreover, the applicability of SbL is not limited to client-interfaces, but

may extend to other parts of the production chain, enabling a unified approach to production control, from studio all the way to consumer interfaces [12].

Bridging the Gap. Tracks and cursors reflect a deep division in media. At a low level, there is a distinction between objects referenced to a timeline and objects with a current value. At a higher level, there is a similar divide between time-driven experiences (e.g., audio, video, and animations) and data-driven experiences (e.g., documents and streams). Many applications attempt to bridge these domains, for instance by introducing interactivity or collaboration into a primarily time-driven experience, or vice versa. However, due to profound platform differences, this often results in complex integrations and ad-hoc solutions. SbL addresses this challenge in low-level programming constructs, allowing bridges to emerge naturally at higher levels, without a steep rise in complexity. Playback and record operations are central in this context, enabling transitions between time-dependent and current state representations as needed.

Data and Control. In time-driven media production, there is an intuitive distinction between data and control. Data refers to media content, the raw material of media production, whereas control refers to actions applied to them, such as activating a source or fading an audio channel. A similar distinction appears in data-driven media, where data might be a file with subtitles and control could be program variables defining playback offset or the visibility of a display element. From this, it may be tempting to conclude that tracks correspond to data and cursors to control. Importantly, though, the *data-control* distinction concerns the *role* of a resource in media production, whereas the *track-cursor* distinction pertains to its *representation*. For example, if a GPS track is displayed in a map overlay, its *(role, representation)* attributes would be *(data, track)*. But, if the same GPS track is played back and used to control the positioning of the map view, the attributes would rather be *(control, cursor)*. Crucially, SbL remains agnostic to the data-control distinction, enabling layering to be applied equally for control and data sources.

8 Conclusion

We imagine a more flexible, web-like model for time-driven media production, where layer orchestration is performed on client devices, from a potentially large set of dynamic data and control sources, yet directed in real-time by a centralized production system. This makes consumer interfaces part of the production system and allows time-driven narratives to be expressed natively on data-driven platforms, through precisely timed interface updates.

This paper introduced *State-based Layering (SbL)*, a conceptual framework in which timing, control, layering, and playback—core concerns of time-driven media production—are addressed within application state, decoupled from data backends and UI. We proposed a methodology where time-driven layering logic is not delegated to media frameworks but expressed in application code, supported by generic programming constructs. *Track* and *Cursor* were introduced

as unifying resource abstractions for entities with *time-dependent* and *current* state, respectively. Moreover, we defined a mechanism in which generic layering operations—such as *record*, *shift*, *merge*, and *playback*—transform and combine live data and control state into renderable output. This mechanism was implemented in *StateLayers*, a JavaScript framework supporting the development of time-driven, layered media experiences in Web apps. Built-in support for time-sensitive, online production control ensures that consumer-side media productions can be directed from a central production context, while layering and rendering is performed consistently across consumer interfaces.

SbL has been validated across multiple levels; *architecture*, *methodology*, *concepts*, *mechanism*, and *framework*. Most notably, this work demonstrates that centralized production control over client-side layering is feasible, and that control, layering, and playback can be fully encapsulated in application state, requiring no changes to backend services or rendering components.

SbL reveals a new class of time-driven applications native to data-driven media platforms. By enabling precise, time-sensitive production control in data-driven consumer interfaces, it presents new opportunities for time-driven media production: with implications for key challenges in the media domain, such as automation, interactivity, adaptation, personalization, and accessibility. Moreover, the utility of SbL extends beyond the traditional media domain, encompassing any system that records real-world events or visualizes data in accordance with timeline progression.

Future Work. SbL requires further validation in real applications and systems, as a guide for refinements in both mechanism and framework implementation. More broadly, SbL may open new research opportunities concerning the role of control and layering in media systems, application design, and/or human-computer interaction (HCI).

Acknowledgements

We acknowledge the use of OpenAI's ChatGPT to improve the quality of the manuscript. All substantive ideas, arguments, and contributions are those of the authors. This work was supported by the Research Council of Norway and MediaFutures (Grant number 323302 and 309339).

A Implications of a State-based Approach

Layering as an independent component. A state-based approach implies that layering can be defined as an independent component, decoupled from both application resources and rendering components. This enables a variety of data and control sources to be used as inputs to layering, and it ensures that layering output can be shared among UI components. Moreover, if resources can be made available under a common abstraction, layering functionality can be reused across different resource types and with different UI components.

Distributed production control. The layering process is driven by changes in data sources or control state. Distributed production control can be achieved if control state and data sources are hosted online. Control actions can then be dispatched from a production system and handled by observing clients. Online control resources may be shared between devices, groups, and across large audiences.

Layering as a state transformation. Layering effectively implements a state transformation, from data and control sources to output state prepared for rendering. It can be considered a pure function (like rendering components) in the sense that outputs are determined solely from inputs at any given moment. The layering function is also time-dependent, implying that layering output ultimately depends on a clock and can change gradually over time. Clocks are regarded as control resources in their own right.

Programmable layering. Layering logic is defined in application code. By leveraging appropriate programming constructs and a state-based programming model, developers can implement complex layering logic tailored to specific application demands. Simple, common-purpose layering functions may also be reused as parts of more advanced or specialized layering functions.

Timing, control and layering. Time-driven media production depends on precise timing in all phases of production, including capture, distribution, and playback. As data-driven platforms generally offer weak support for timing, these requirements must be addressed by the layering, ensuring that internal time relations are preserved from capture to rendering. This applies equally to data sources and control state.

B Prior Research

State-based Layering (SbL) represents the culmination of an extensive research track into time-driven and data-driven media systems and applications. Our earlier work addressed the challenges of time-control and synchronization in multi-device media applications, resulting in the development of generic concepts such as MediaStateVector [13], TimingObject [30], MediaSync [17], and Sequencer [10]. These concepts collectively informed the development of Timingsrc [31], a reference implementation and programming model for web-based,

time-driven media applications. SbL extends this model by providing a unifying resource abstraction for control and data sources alike, yielding a more flexible model with improved support for composition and customization.

More recently, our research has focused more directly on the role of control in media systems. StateTrajectory [11] conceptualized control in media not merely as a real-time signal, but as a persistent online-hosted resource, thereby blurring the classical distinction between production control and interactive presentation control. StateTrajectory also recognized the need to treat control as a media object in its own right, with independent support for recording, time-shifting and playback. Tracks and cursors were first introduced in this context, although primarily as an implementation detail. SbL builds directly on this foundation, promoting tracks and cursors as first class concepts, rooted in the fundamental distinction between time-dependent state (tracks) and current-state (cursors).

Our most recent publication, Control-driven Media (CdM) [12], explored the broader implications of this research. It introduced a highly flexible media model in which media experiences are assembled from independent, online-hosted control and data sources, with the assembly process becoming increasingly individualized along the production path, from studio to consumer interfaces. However, this work did not specify a specific mechanism for assembly. SbL is our proposed solution for this challenge.

References

1. Adobe: Adobe After Effects. Visual effects and motion graphics software (2025). Accessed online: https://www.adobe.com/products/aftereffects.html
2. Adobe: Adobe Animate (2025). Accessed online: https://www.adobe.com/products/animate.html
3. Adobe: Adobe Photoshop (2025). Accessed online: https://www.adobe.com/products/photoshop.html
4. Adobe: Adobe Premiere Pro (2025). Accessed online: https://www.adobe.com/products/premiere.html
5. Apple: Final Cut Pro (2025). Accessed online: https://www.apple.com/final-cut-pro/
6. Apple: Logic Pro. Professional music production software (2025). Accessed online: https://www.apple.com/logic-pro/
7. Apple: UIKit Framework reference (2025). Accessed online: https://developer.apple.com/documentation/uikit
8. Armstrong, M., Brooks, M., Churnside, A., Evans, M., Melchior, F., Shotton, M.: Object-based broadcasting-curation, responsiveness and user experience. In: IBC2014 Conference. IET Digital Library (2014). DOI 10.1049/ib.2014.0038
9. Arntzen, I.M.: StateLayers: State-driven Layering implemented in JavaScript (2025). Accessed online: https://github.com/ingararntzen/state-layers
10. Arntzen, I.M., Borch, N.T.: Data-independent Sequencing with the Timing Object: A JavaScript Sequencer for Single-device and Multi-device Web Media. In: Proceedings of the 7th International Conference on Multimedia Systems, MMSys '16, pp. 24:1–24:10. ACM, New York, NY, USA (2016). DOI 10.1145/2910017.2910614
11. Arntzen, I.M., Borch, N.T., Andersen, A.: State Trajectory. A Unifying Approach to Interactivity with Real-Time Sharing and Playback Support. In: K. Arai (ed.) Proceedings of the Future Technologies Conference (FTC) 2023, Volume 2, pp. 1–20. Springer Nature Switzerland, Cham (2023). DOI 10.1007/978-3-031-47451-4_1
12. Arntzen, I.M., Borch, N.T., Andersen, A.: Control-Driven Media: A Unifying Model for Consistent, Cross-platform Multimedia Experiences. International Journal of Advanced Computer Science and Applications (IJACSA) **15**(9) (2024). DOI 10.14569/IJACSA.2024.0150904. URL http://dx.doi.org/10.14569/IJACSA.2024.0150904
13. Arntzen, I.M., Borch, N.T., Needham, C.P.: The Media State Vector: A Unifying Concept for Multi-device Media Navigation. In: Proceedings of the 5th Workshop on Mobile Video, MoVid '13, pp. 61–66. ACM, New York, NY, USA (2013). DOI 10.1145/2457413.2457427
14. Avid: Pro Tools. Professional audio recording and music production software (2025). Accessed online: https://www.avid.com/pro-tools
15. BBC Research & Development (R&D): Object-based Media (2015). Accessed online: https://www.bbc.co.uk/rd/object-based-media
16. Blender: Blender. Free and open source 3d creation suite (2025). Accessed online: https://www.blender.org/
17. Borch, N.T., Arntzen, I.M.: Mediasync Report 2015: Evaluating timed playback of HTML5 media. Tech. Rep. 28, Norut Northern Research Institute (2015). URL https://hdl.handle.net/11250/2711974. https://hdl.handle.net/11250/2711974
18. EaseLive: Interactive tv graphics platform for live sports and consumer-centric streaming. (2025). Accessed online: https://www.easelive.tv/

19. Evans, M., Ferne, T., Watson, Z., Melchior, F., Brooks, M., Stenton, P., Forrester, I., Baume, C.: Creating object-based experiences in the real world. SMPTE Motion Imaging Journal **126**(6), 1–7 (2017)
20. Facebook: React. The library for web and native user interfaces (2013). Accessed online: https://react.dev/
21. Gimp: GIMP — GNU Image Manipulation Program (2025). Accessed online: https://www.gimp.org
22. Google: Google Maps (2005). Accessed online: https://www.google.com/maps
23. Google: Angular is a web framework that empowers developers to build fast, reliable applications. (2016). Accessed online: https://angular.dev/
24. Google: View and Viewgroup in Android (2025). Accessed online: https://developer.android.com/reference/android/view/ViewGroup
25. Harris, R.: Svelte - web development for the rest of us (2016). Accessed online: https://svelte.dev/
26. OpenLayers: OpenLayers: A high-performance, feature-packed library for all your mapping needs (2006). Accessed online: https://openlayers.org
27. Unity: Unity real-time development platform (2005). Accessed online: https://unity.com
28. Unreal: Unreal engine (1998). Accessed online: https://www.unrealengine.com
29. W3C: Web Audio API. Tech. rep., World Wide Web Consortium (W3C) (2021). Accessed online: https://www.w3.org/TR/webaudio/
30. W3C Multi-device Timing Community Group: Timing Object; Draft Community Group Report. Tech. rep., W3C (2015). Accessed online: http://webtiming.github.io/timingobject/
31. W3C Multi-device Timing Community Group: Timingsrc. Multi-device Timing for the Web (2015). Accessed online: https://webtiming.github.io/timingsrc
32. WHATWG: DOM Living Standard. Tech. rep., WHATWG (2025). Accessed online: https://dom.spec.whatwg.org/
33. WHATWG: HTML Living Standard — The `<track>` element. Tech. rep., WHATWG (2025). Accessed online: https://html.spec.whatwg.org/multipage/media.html#the-track-element
34. You, E.: Vue.js - The Progressive Javascript Framework (2014). Accessed online: https://vuejs.org/