Workflow Support for Live Object-Based Broadcasting

Jack Jansen
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands
Jack.Jansen@cwi.nl

Pablo Cesar
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands
Technische Universiteit Delft
Delft, the Netherlands
Pablo.Cesar@cwi.nl

Dick Bulterman
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands
Vrije Universiteit
Amsterdam, the Netherlands
Dick.Bulterman@cwi.nl

ABSTRACT

This paper examines the document aspects of object-based broadcasting. Object-based broadcasting augments traditional video and audio broadcast content with additional (temporally-constrained) media objects. The content of these objects – as well as their temporal validity – are determined by the broadcast source, but the actual rendering and placement of these objects can be customized to the needs/constraints of the content viewer(s). The use of object-based broadcasting enables a more tailored end-user experience than the one-size-fits-all of traditional broadcasts: the viewer may be able to selectively turn off overlay graphics (such as statistics) during a sports game, or selectively render them on a secondary device. Object-based broadcasting also holds the potential for supporting presentation adaptivity for accessibility or for device heterogeneity.

From a technology perspective, object-based broadcasting resembles a traditional IP media stream, accompanied by a structured multimedia document that contains timed rendering instructions. Unfortunately, the use of object-based broadcasting is severely limited because of the problems it poses for the traditional television production workflow (and in particular, for use in live television production). The traditional workflow places graphics, effects and replays as immutable components in the main audio/video feed originating from, for example, a production truck outside a sports stadium. This single feed is then delivered near-live to the homes of all viewers. In order to effectively support dynamic object-based broadcasting, the production workflow will need to retain a familiar creative interface to the production staff, but also allow the insertion and delivery of a differentiated set of objects for selective use at the receiving end.

In this paper we present a model and implementation of a dynamic system for supporting object-based broadcasting in the context of a motor sport application. We define a new multimedia document format that supports dynamic modifications during playback; this allows editing decisions by the producer to be activated by agents at the receiving end of the content. We describe a prototype system to allow playback of these broadcasts and a production system that allows live object-based control within the production workflow. We conclude with an evaluation of a trial using near-live deployment of the environment, using content from our partners, in a sport environment.

CCS CONCEPTS

• **Applied computing → Markup languages**; *Multi / mixed media creation*; • **Information systems →** *Markup languages*;

KEYWORDS

Object based video, Declarative languages

ACM Reference Format:
Jack Jansen, Pablo Cesar, and Dick Bulterman. 2018. Workflow Support for Live Object-Based Broadcasting. In *DocEng '18: ACM Symposium on Document Engineering 2018, August 28–31, 2018, Halifax, NS, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209280.3209528

1 INTRODUCTION

When the SMIL language was initially released in 1998, one of the hopes of the development team was that it would facilitate bringing TV content to the Web [14]. By introducing a declarative container document that held references to networked multimedia, the hope was that complex content streams could be bundled together based on end-user preferences, ushering in a new era of on-demand content delivery. In order to meet the needs of a diverse set of multimedia content producers and consumers, SMIL (and other languages being developed at the time, such as NCL) had extensive support for temporal hierarchies, timing graphs, and linking architectures, as well as visual structures that supported dynamic content layout and sizing. Content control facilities were available that allowed content to be tailored to the network connection characteristics and various local parameters, such as screen size or preferred language. There were pro's and con's to each language developed, but in hindsight they all shared a common problem: they were far too complex to meet the workflow, transmission and rendering requirements of 'real' broadcast television production.

While many multimedia researchers may have been over-ambitious in defining adaptive formats, many in the area of television technology never transcended a content production model that effectively resulted in the 'burning' of content in fixed, producer-defined positions in an audio-video stream. Within the world of television production, controlling the visual message remains of paramount importance, at the expense of client-based customizations or even the flexible sharing of (parts of) the content stream among the set of primary and secondary screens that are at the disposal of a modern media viewer.

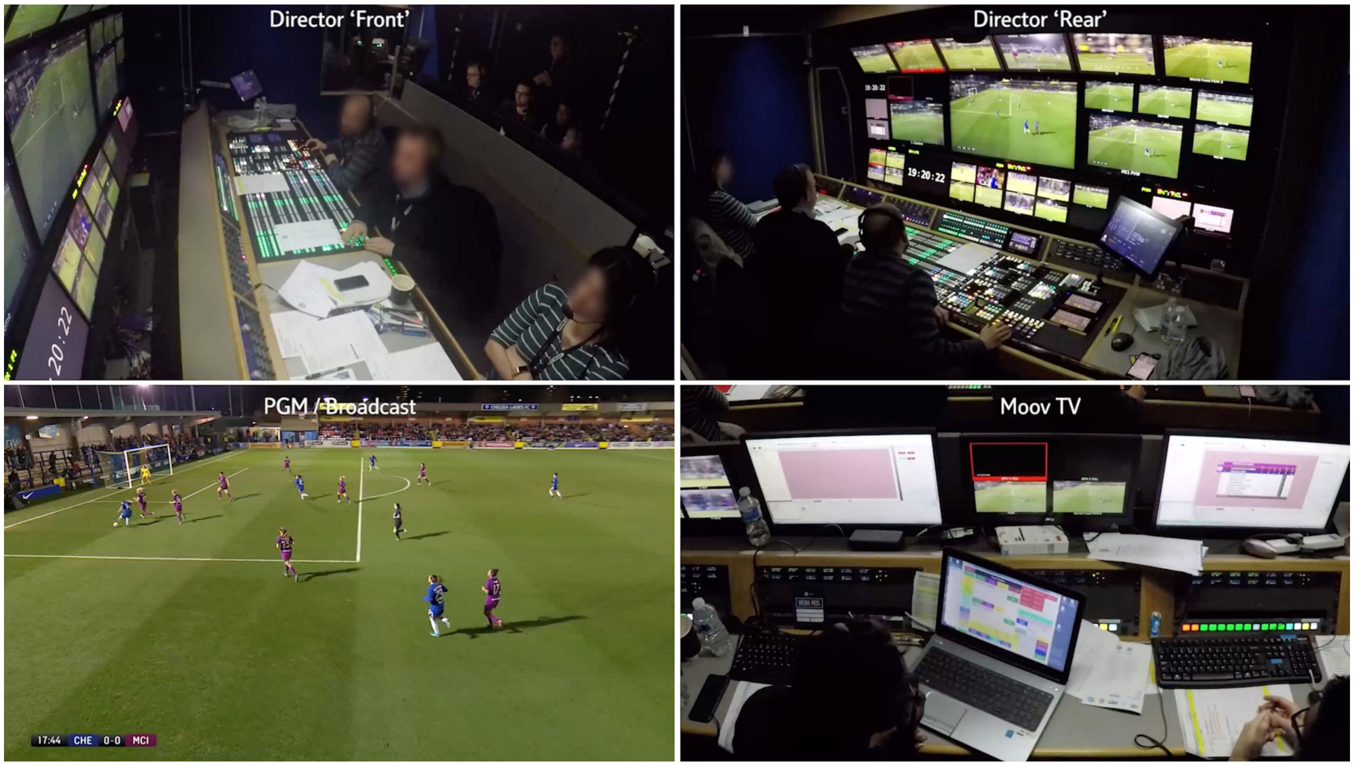
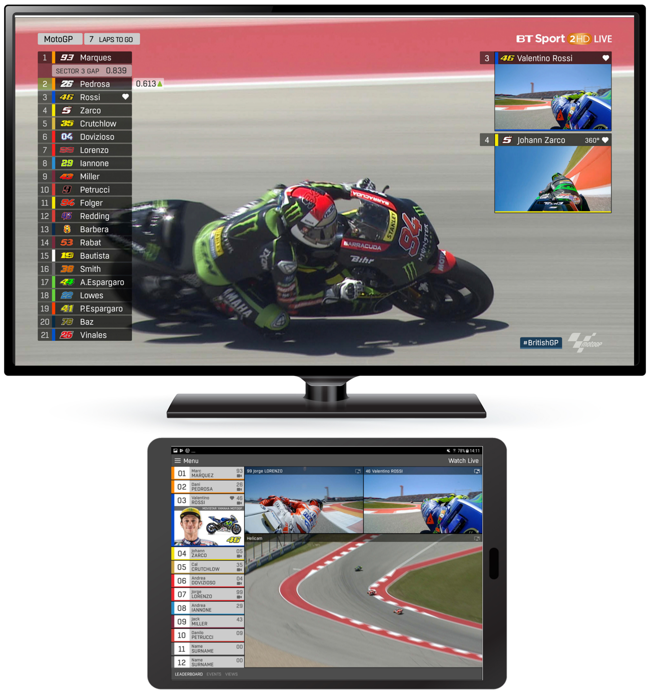


Figure 1: A Typical OB van as used for producing a sports broadcast. (image courtesy BT Sport)

This paper reports on the development of a document model that has been developed to meet the production workflow needs of the television community, while still offering the end-user a great degree of flexibility in controlling the selection of on-screen content. Our work, which has been developed in a partnership with European broadcasters, content distributors and other related parties, has been focused on the integration of object-based broadcasting of multiple content components in the context of a sports-television application.

Figure 1 provides a view of the typical production setting in an OB (outdoor broadcasting) van. We see three sets of activity, including the director's station and video mixer at top left and right and the graphics integration console at bottom. The ultimate video feed coming from the truck is shown bottom left. The content stream that is created in the van is distributed via a multi-stage process to the homes of (paid) viewers. The end-to-end delay from action on the field to consumption on the couch can vary from 10-50 seconds, depending on the architecture used.

The work reported in this paper has concentrated on tests using content produced for motor-sports: the MotoGP. Figure 2 shows a conventional screen at the top, containing the main video of the race and a set of content overlays. One of the goals of our work has been to define a light-weight and flexible timing model that can be used as the basis of supporting flexible end-user consumption across a primary and secondary screen (shown in the bottom of Figure 2.) This work touches on many aspects of content selection, scheduling, transmission, delivery and routing within the home.



MotoGP		7 LAPS TO GO
1	93	Marques
	SECTOR 3 GAP	0.839
2	26	Pedrosa 0.613
3	46	Rossi
4	5	Zarco
5	35	Crutchlow
6	04	Dovizioso
7	99	Lorenzo
8	29	Iannone
9	43	Miller
10	9	Petrucci
11	94	Folger
12	45	Redding
13	8	Barbera
14	53	Rabat
15	19	Bautista
16	38	Smith
17	44	A.Espargaro
18	22	Lowes
19	41	P.Espargaro
20	76	Baz
21	25	Vinales

Figure 2: Object-based viewing of a MotoGP race. (from [9])

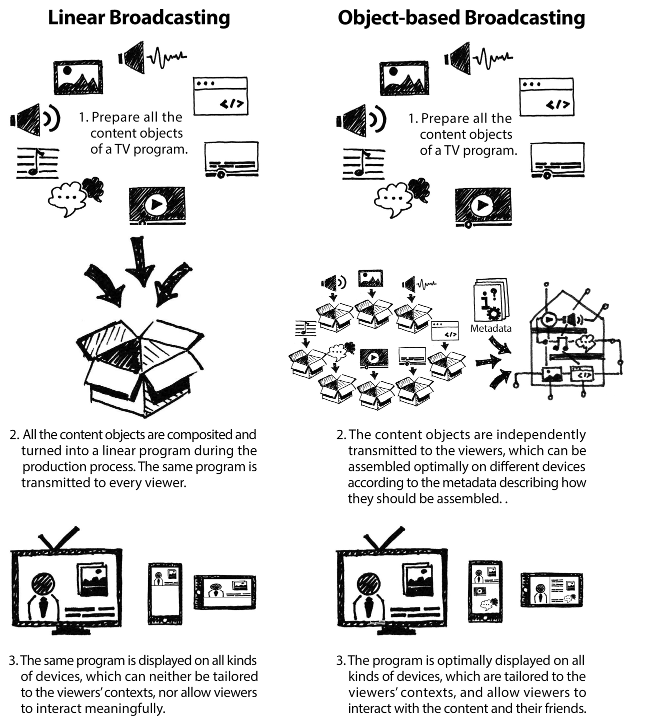

Linear Broadcasting
Object-based Broadcasting
1. Prepare all the content objects of a TV program.
1. Prepare all the content objects of a TV program.
Metadata
2. All the content objects are composited and turned into a linear program during the production process. The same program is transmitted to every viewer.
2. The content objects are independently transmitted to the viewers, which can be assembled optimally on different devices according to the metadata describing how they should be assembled. .
3. The same program is displayed on all kinds of devices, which can neither be tailored to the viewers' contexts, nor allow viewers to interact meaningfully.
3. The program is optimally displayed on all kinds of devices, which are tailored to the viewers' contexts, and allow viewers to interact with the content and their friends.

Figure 3: Object-based broadcasting compared to linear broadcasting. (from [9])

Most of the details of the end-to-end chain are beyond the scope of this paper, but will be addressed in passing in the text below. Our focus is on defining a single cloud-based presentation document that can be shared with potentially millions of viewers simultaneously. The creation process of this document allows a broadcaster to identify and schedule information objects (such as replays, statistics summary, scoreboards, background information) in such a manner that the broadcaster retains full control of content, but which still enables viewer-side adaptation and control across a local set of devices. While our solution has the potential to work for both live and on-demand content, the operational constraint in our research was to support live broadcasts only.

The main contribution of this paper is a structured temporal document format and the accompanying editing semantics. This is supported by an implementation, and by demonstrating that this implementation fits the workflow for live editing of object based video.

Our paper is structured as follows. Section 2 summarizes the document requirements for supporting live object-based broadcasting. Section 3 reviews related work. Section 4 discusses the general design of our system and Section 5 considers the particular timeline document format that we propose. Section 6 discusses our implementation and Section 7 our experimental evaluation. We close with conclusions and issues for future research.

2 SYSTEM REQUIREMENTS

The general process of conventional linear broadcasting and the emerging art of object-based broadcasting are illustrated in Figure 3. At left, we see how linear broadcasting uses a fixed packaging model to collect and arrange media objects for distribution to a wide range of consumer devices. Content control is strictly managed by the producer, with the consumer having little influence on how and where this content is adapted on the desired client device. At right, we see a sketch of the object-based model. The individual content objects are created using a familiar (to the broadcaster!) workflow, and then individually packaged along with scheduling meta-data for storage on an intermediate content delivery network (CDN). Using information on the local devices, the content can be adapted for placement on a home device. Note that the content itself remains under full control of the producer, but by being unbundled, it can be tailored to the needs of the user.

In our work, the ‘meta-data’ in Figure 3 is a structured document model that contains a presentation timeline description. This description has a number of special characteristics that make it suitable for object based broadcasting. First and foremost, the document structure is simple, containing only limited time references to other objects. This allows content to be easily inserted and deleted from the document without forcing a complete re-parse of the document.

The document format has been designed to assist in meeting the following constraints across the system:

- It must allow for the seamless playback of content at the viewer side, providing a “baked-in content” experience;
- It must allow viewer-side rendering and layout control of content across home devices;
- It must allow synchronized playback of content on multiple associated devices on in the home;
- It must support the inclusion of live events at the producer side and scheduled rendering of these events at the client;
- It must support flexible entry and exit of devices, while respecting the ‘live’ model of the broadcast; and
- It must hold the potential for non-live, on-demand playback of the presentation document in a way that respects the original scheduling of content objects.

We note that not all of these requirement are new, but neither of the two main declarative formats that have historically addressed these issues (NCL and SMIL) have the facilities to incrementally grow or shrink a single shared instance in a manner that supports live content editing. We return to these requirements in the relevant sections below.

3 RELATED WORK

This work is part of a larger collaboration between different parties on designing, developing and deploying an end-to-end object-based broadcast chain. This paper emphasizes the document-oriented aspects of our work, we refer to companion papers by Li [9] for a treatment of the production workflow analysis and by Walker [15] and Kegel [8] for a more in-depth discussion of the system architecture.

There are a number of case studies on delivery and production of object-based video. Ursu et al [13] describe design and deployment of a system that allows viewers to influence the storylines of

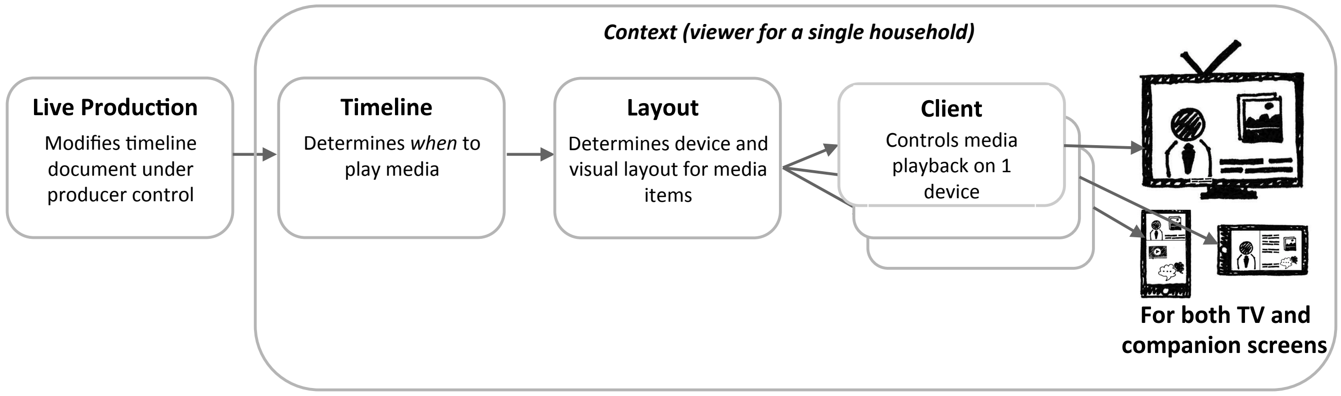

Context (viewer for a single household)
Live Production
Modifies timeline document under producer control
Timeline
Determines *when* to play media
Layout
Determines device and visual layout for media items
Client
Controls media playback on 1 device
For both TV and companion screens

Figure 4: Control flow for object-based video chain.

television programs in various different genres. Cake [4] treats a single genre, cooking programs, but adresses more of the whole production workflow. Squeezebox [1] is an example of ongoing work to look into production tools for object-based video.

That it is in principle possible to modify the timegraph of a declarative multimedia document during playback is shown for NCL by de Resende Costa [5]. That it is in principle impossible in the general case is shown for SMIL by the authors [7]. Our current work leans heavily on these two papers and extends the work therein. We show that it is possible to create a declarative document format that is “SMIL-like” in design but that can be modified during playback. These two papers also include ample references to adaptive multimedia documents in general, which we will not repeat here.

Two standardization efforts that are related to our object-based video delivery approach. MPEG-TEMI [10] is a standard for encapsulating references to external objects and associated timing into an MPEG-2 Transport Stream. This enables scenarios in which additional extrinsic media objects can be rendered synchronously with the television broadcast [2]. The references to the external media are transmitted in-line in the transport stream, which contrasts with our out-of-line method that facilitates companion devices. Our solution also aims at adaptability and user interaction, which MPEG-TEMI does not handle.

MPEG-MORE [11] is a draft standard that aims at providing mechanisms to orchestrate in a coherent manner media experiences that have multiple media sources. It has a richer model than TEMI and a wider reach, also targeting multiple sources and sinks of information. Like our solution MPEG-MORE uses DVB-CSS [6] for low level synchronization between devices. MPEG-MORE does not address user interaction and adaptation currently.

4 SYSTEM DESIGN

The overall control flow of the whole system is sketched in Figure 4. There is a single instance of the *Live Production* box, which is the component that allows the producer to issue his commands to trigger visuals, effects and additional media to be played out.

In a true live setting there will be an individual *Timeline* and *Layout* component for each household. Therefore, there can potentially be millions of these and these components have been implemented as cloud services. For each device (television or settop box, companion device) in each household there would be a *Client* component, which will run on the device it controls. The set of timeline, layout and clients for a single home we call a *context*, and within a single context all media playout is coordinated. In addition to a context for every household there is one more context active: the *preview player context* that allows the producer to see the visual effect of his editing commands.

The timeline component determines the start and stop times of individual media items, relatively to the base timeline (the main video broadcast feed). It issues commands to the layout component, which determines on which device, if any, this media item needs to be played out, and where it should be placed on the screen. The layout component forwards the start and stop commands to the correct client instance, which now simply has to play the media at the right time.

Individual handheld devices can join and leave the context at will, by contacting the layout component, which de facto acts as the context controller. The layout component remembers all media items for which a start has been received and no stop yet, and issues the correct set of commands to handhelds that join during the playback session.

Within a context feedback flows in the opposite direction (right to left in Figure 4): clients report play position of the individual media items (including the main feed) to the layout component and hence to the timeline component. This enables the timeline component to determine the current play status and time position of each media item.

The timeline, layout and clients for a live broadcast are the same as those used for a prerecorded broadcast, as sketched out in [8]. The only difference is that in a prerecorded broadcast the timeline component plays out a static document with all media items and timing determined beforehand, whereas in the live situation that document is modified on the fly, by the live production component issuing modification commands to the document. The live production tool simply issues a single edit operation to all timeline components. As all timeline components are playing the same document this edit operation will therefore have the same effect in all households.

The preview player context is identical to a normal context, with two exceptions:

- Its timeline component sends feedback on media playout status and time position to the live production component. This allows the production component to give visual feedback to the producer (in the form of lighting up buttons as components are playing and such), but more importantly if gives the live production component itself a sense of the play status of the document, so that it can insert media items with the correct timing information.
- In a true live setting the preview player context will play a timeline document that uses a version of the main video feed that is a few seconds ahead of the normal broadcast feed. This is normal practice for live television broadcast, as the video feed will have to pass through encoders, encrypters and distribution infrastructure, and possibly be monitored for inappropriate content such as nudity or bad language. For our setting this has the distinct advantage that our realtime requirements become less stringent: we have a few seconds to spare to forward the edit decisions of the producer to the timeline components for the home viewers.

In terms of the requirements defined in Section 2, it is the client that has the responsibility for ensuring seamless playback based on the elements in the timeline document. The layout components supports cross-device placement and adaptation, to the extent allowed by the content owner. The processing of the document occurs via the layout module, which also detects and enforces rendering of objects that have been inserted during the live editing process. Note that the layout component also enforces the constrained entry/departure of new devices within the context of rendering in the home to ensure enforcement of the live (fixed wallclock) broadcast model. At the same time, it is also the layout component that can be used for on-demand access, once this is allowed by the content owner.

5 TIMELINE DOCUMENT

5.1 Design Principles

The document format design has been guided by the requirements in Section 2. The primary external representation format is XML with namespaces (which we use in this paper), but the format has been designed to also be representable in JSON, and in-core data structures can follow the document format closely.

The format shares a number of requirements with SMIL [3]:

(1) Hierarchical temporal containment: the timing aspects of an element (when it is active) is fully contained in the timing aspects of its parent.
(2) Parallel and sequential composition: the basic operations that govern timing relationships are elements that activate their children either in parallel, playing out at the same time, or sequentially, playing out one after the other in document order.
(3) Automatic inference of timing and synchronization: elements without intrinsic timing aspects (such as images) should pick up their timing from their parent, or indirectly from their siblings.

But only these three aspects were taken from SMIL. Three more requirements for the new format are in conflict with SMIL:

(4) Timing aspects of each element are governed strictly by the element itself, its parent and its direct children. This leads to a format where all scheduling and synchronization can be computed on a fairly local level, based only on the state of the element and its direct neighbors in the tree. As a result, constructs like SMIL timebase which enables synchronization relationships outside the tree hierarchy, are not included.
(5) Timegraph equivalence. The timegraph that is used in memory while playing the document must be the same as the external representation, with some annotations on each element to encode the current state of that element. SMIL (and many other declarative multimedia formats such as NCL[12]) needs to transform the external document graph into an internal timegraph with a different structural representation to facilitate execution.
(6) Every temporal operation, rule or modifier is encoded as an individual element type. Many formats – including SMIL – encode things as attributes, but this makes it difficult to specify how features interact, often requiring a large amount of specification text. The description of how dur, repeatDur and repeatCount interact in SMIL can be seen as an example of this issue. By encoding each feature as an individual element it becomes a lot easier to specify the semantics: the nesting inherently specifies the interplay.

The largest difference between our document format and SMIL is that many aspects of SMIL are out of scope for this format (layout, SMIL state, transitions and animations). Also, SMIL had the goal of human-readability and conciseness that this format does not share. Such a goal would be in conflict with requirements (4) through (6): these requirements are essentially about decomposing functionality so the effect of document modification operations can be specified. A conciseness goal would go in the opposite direction of combining functionality for authoring convenience.

The timegraph equivalence requirement (5) of our format ensures that our documents can in principle be edited at runtime. In [7] the authors show that the SMIL language features in the Structural Cluster (their term) are the only features that cannot be modified, and with timegraph equivalence the structural cluster is essentially empty for our format. The local timing requirement (4) ensures that this editing can also be implemented in a practical way: recomputation of timing aspects can be done on a local level without taking the whole document timegraph state into account.

While we have not completed the full analysis of the timing model for live/on-demand access, we believe that requirement (6) will allow us to specify the full temporal semantics of our format in a concise way, we aim at something that can be fully explained in 10 pages of text (as opposed to the hundreds of pages of the

```
<tl:par tl:endsync="all">
    <tl:ref url="video.m4v"/>
    <tl:ref url="caption.txt"
            tl:fill="freeze"/>
    <tl:ref url="flash.jpg"
            tl:fill="remove"/>
</tl:par>
```

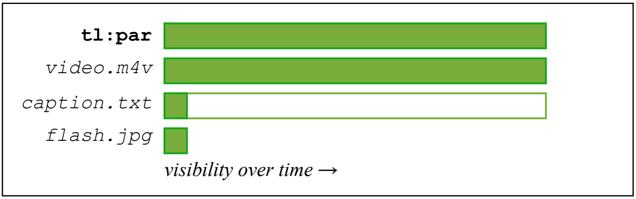


Figure 5: Example of inferred timing

SMIL specification [14]). It may also be possible to formally specify the semantics using an appropriate formalism, something that has never been possible for the full SMIL timing model.

We will now first outline the document structure elements, and then proceed to show editability.

5.2 Format

The timeline document, and hence the timegraph, consists of basic elements (leaf nodes in the document tree), grouping elements, (interior nodes in the tree) and modifying elements (interior nodes in the tree with only a single descendant). In this section we will use the XML namespace prefix `tl:` to indicate elements and attributes that are relevant to the timeline document format.

The most basic elements in the timeline documents are the `<tl:ref>` elements that play media or have other external side-effects such as changing layout or playback parameters. There can be multiple types of these, but from the timeline perspective they are all semantically identical and go through a number of discrete steps:

(1) They are created and initialized by the timegraph execution engine,
(2) Then they are started by the timegraph execution engine,
(3) They may then end of their own accord because the underlying media has finished playing,
(4) They are stopped by the timegraph execution engine (either before or after the previous step),
(5) They are destroyed by the timegraph execution engine.

Here, and in the rest of this document, we use the words *finished* and *stopped* with precise and distinct meaning. *Finished* means that the media file or element has reached its end. *Stopped* means that timegraph execution engine has instructed the media playback component or the element to stop. Finished is therefore a change of playback state of an element, which may result in state changes in the timegraph. Stopped is the reverse: a state change probably caused by state changes in the timegraph.

As stated, there can be multiple types of elements that play media, and they can have many attributes to control that playback. And, indeed, for the 2-IMMERSE use case there is an additional element type `<tim:update>` which behaves exactly like `<tl:ref>` from a timeline aspect but in stead of playing media it changes parameters on already running media items. For the remainder of this section we will group all these elements under the `<tl:ref>` moniker. The only interesting attributes are `tl:fill` and `tl:prio`, which are discussed below.

There are two more basic elements that are very similar to `<tl:ref>`, except they do not have the side effect of playing media:

- `<tl:sleep tl:dur="20s"/>` does nothing and finishes playing 20 seconds after it is started.
- `<tl:wait tl:event="userevent"/>` does nothing and finishes playing after the named event has been received from a player (client component) through user interaction or a system event.

From requirements (1) through (3) follow two types of grouping elements, representing parallel and sequential temporal composition of their children. Sequential playback is implemented by `<tl:seq>`, which simply executes its children in order: when the `<tl:seq>` is started its first child is started, which it finishes the next child is started and so on. When the last child has finished playback the `<tl:seq>` itself finishes playback.

Parallel composition is done with the `<tl:par>` element. All children are started when the `<tl:par>` element starts, but the stopping of elements and the finishing of the `<tl:par>` element itself is a bit more involved and governed by attributes on the element itself and its children:

- The `tl:end` attribute on the `tl:par` element determines when the parallel composition element finishes. The values can be a child identifier (`xml:id`) or one of the values first, all or master. A child identifier means the par finishes when that specific child finishes playback. *First* means the par finishes when any child finishes, *all* means it finishes only when all of its children have finished and *master* means that one specific child determines when the par finishes, but which child is determined by the priorities of the children. The highest priority child element will govern the par timing.
- The `tl:prio` attributes on the children are used to determine the priorities, but these priorities can be modified at runtime depending on the type of media the child element is playing back, with timed media (audio, video) having a higher default priority that static media items (images, text).

```
<video begin="5s" dur="5s" src="video.m4v"/>
```

```
<tl:par tl:end="first">
  <tl:seq>
    <tl:sleep tl:dur="5s"/>
    <tl:ref url="video.m4v"/>
  </tl:seq>
  <tl:sleep tl:dur="10s"/>
</tl:par>
```

Figure 6: SMIL (top) and Timeline format (bottom) equivalence.

- The `tl:fill` attribute on a child element determines what happens when that child element finishes playback, but the par parent element has not finished yet. If the value is remove the child element will be stopped (removing it from the screen), but if the value is freeze the stop will be delayed until the `<tl:par>` element itself finishes. This effectively keeps media items visible. Figure 5 shows an example of how this can be used to have media items, especially static media items such as pictures, pick up the duration from other parallel media items.

Note that the `tl:prio` and `tl:fill` attributes are not only used on leaf elements but also on interior elements. This means that, for example, a `tl:seq` with `tl:fill="freeze"` that has a last child that also has `tl:fill="freeze"` will effectively extend the visibility of that last child until the `tl:seq` itself is stopped.

There are two modifying elements. The first one is `<tl:repeat>`, which has a `tl:count` attribute and simply runs its child element that many times: when the repeat is started it starts its child, when the child finishes it is stopped and immediately started again, and this continues count times.

The other modifying element is `<tl:conditional>` which has a `tl:expr` attribute. When the conditional element is started it checks the expression for being *true* or *false*. The expression will usually refer to a condition set by the user or the playback system (such as whether the user prefers a specific language, or has a screen of a specific minimum size). If the condition is true the child element is played and the conditional element finishes when the child element does. If the condition is false the element simply finishes directly.

Most SMIL semantics can be expressed in this new language, but often the the new constructs are much more verbose. See Figure 6 as an example: to start playback of a video after a delay and stop it after playing back 10 seconds our timeline format needs 7 lines what would require only a single line in SMIL. Many constructs in SMIL will require such a nested `tl:par`/`tl:seq` composition with `tl:sleep`s in there. As another example you can think of how the equivalent of SMIL `repeatDur` would be implemented.

The verbosity is caused by our new requirements (4) through (6) and we feel it is a small price given the benefits those requirements entail.

Some temporal features of SMIL are currently not implementable in our timeline format, notably `<excl>`, `<switch>`, multiple begin and end times and syncbases. With the exception of syncbases we could add elements for these features to the format in the future, when needed.

5.3 Modification Operations

We have identified the following atomic edit operations, basically all XML DOM operations, and we believe all possible meaningful document transformations can be decomposed into a sequence of these:

- Adding a new child subtree to an existing (`tl:par` or `tl:seq`) parent element.
- Removing a child subtree from an existing (`tl:par` or `tl:seq`) parent element.
- Inserting a new (`tl:par` or `tl:seq`) parent around an existing element.
- Deleting a (`tl:par` or `tl:seq`) parent around a single child.
- Changing an attribute.

The semantics of all insertion or removal operations depend on whether the new parent element (or old parent element for removal) is currently active or not. If the parent is not active we simply add the new element in inactive state or remove it.

Adding a child to an existing active `tl:par` parent will cause that parent to recompute its state, basically repeating the algorithm on its `tl:end` attribute and the `tl:prio` and `tl:fill` attributes of all its children (including the new one). This will result in a new state for the `tl:par` itself and each of its children. For each child for which the new state does not match the old state it issues the start and stop commands, which will most likely include a start command for the new child and a command to advance the clock of the new child to the correct position.

Adding a child to an existing active `tl:seq` parent depends on where the child is added, with respect to the currently active child. If the new child is added after the active child it is simply added in inactive state. If it is added before the active child things are more complicated. Each child that has been started and stopped remembers how long its duration was, and the `tl:seq` element knows its current (old) clock value. The active element is stopped, the new element is started and the `tl:seq` element sets its clock to the clock value at which that new element would have started if it had been in the document from the beginning. Now a clock advance instruction is sent to the new element, to try and fast forward to the old clock value. This procedure is repeated with successive child elements until the `tl:seq` is back at its old clock value, at which time normal playback continues.

Removing a child from an existing `tl:par` or `tl:seq` parent basically runs the same algorithms as for adding of an element.

Inserting a parent around an existing element initializes the state of the new parent (whether it is active or not, its clock value if it is active, and possibly its last run duration) to the same values as the child element.

Deleting a parent around a single child is a no-op, except possibly for implementation details such as updating the administration on the grandparent.

Changing an attribute depends on which attribute is changed:

- For `tl:dur`, `tl:event` and `tl:count` the element updates its internal administration.
- For `tl:end` the `tl:par` element runs the same algorithm as for child insertion or removal.
- For `tl:prio` and `tl:fill` the parent of the element on which the attribute is changed runs the same algorithm as for child insertion or removal.
- For all other attributes nothing happens with respect to the timegraph, but if the element is active the attribute change may need to be forwarded to the client playout component (so that it can change the media that is played back, for example).

6 IMPLEMENTATION

6.1 Global Structure

The project in which this work was undertaken aims at being close to market, and therefore scalability and deployability are issues that we need to address. The full architecture is described elsewhere [8, 15], here we will only give a global overview.

Referring back to the diagram of Figure 4, our layout and timeline components and the bulk of the live production tool are implemented as lightweight REST services and deployed using cloud infrastructure. Currently this runs in the AWS cloud but is portable to other cloud infrastructures, and it has actually been deployed to other cloud providers in the past. There are numerous other services in our architecture, for onboarding, authentication, distribution of CENC media decryption keys, logging and many other features that fall outside of the scope of this paper.

The client components are basically Javascript webapps that communicate with the cloud services using REST and websockets, but they need some HbbTV 2.0 functionality. Specifically, they need DVB-CSS to enable the handheld devices to discover the settop box or television, so the devices can discover the context to which they belong. DVB-CSS is also used to synchronize the clocks and media positions within the context, so all devices play out synchronously. For the handheld devices DVB-CSS is implemented in the Cordova application framework [1] in which the javascript runs. As no feature-complete HbbTV 2.0 television sets are on the market yet we have implemented the TV client on a small PC (Intel NUC) running a handcrafted Linux installation with a Chrome web browser in kiosk mode and a Python implementation of DVB-CSS[2].

The computational load on the clients is minimal, due to the fact that all coordination is done by timeline and layout components, and the clients could be implemented in settop boxes and internal TV processors. The one computationally intensive operation done by the client is video decoding and decrypting and TVs will have dedicated hardware support for this.

The front end of the live production tool is currently a normal web application using REST and websockets to communicate with the production tool backend. The API is such that multiple frontends can be used to work on the same live production simultaneously with multiple people.

The front end provides the producer with a list of commands that are specific to this broadcast, such as "show the channel logo", "hide the channel logo" or "show a replay from hh:mm:ss.ff with a duration of n seconds", where the time position and duration are parameters the producer can enter. The current list of available commands is obtained from the backend, and will vary during the broadcast. For example, replays are only available during the race and not during the introductory section, and hiding the logo is only available after it has been shown first.

As the producer selects commands the frontend communicates these to the live production backend. The backend translates these into XML edit operations, with the correct timing information inferred from the current time position of the main video feed. These XML edit operations are then forwarded to all timeline components, which modify their documents, which causes the right content to show up in the viewers' homes.

In the live setting (and for the preview player in the near-live setting) the initial timeline document is served to the timeline component from the production tool backend. This document contains a generation number, initially zero and incremented by one for each edit operation. When a new context joins the timeline component gets a recent version of the document and if it notices it has missed edit operations (because of a mismatch in edit operation number and document generation number) it will request the missing operations from the production tool backend. With this mechanism all contexts (and the production tool backend) share the same view of the document.

6.2 Timeline Execution

The implementation of timeline component is a REST-based web service in Python. It has not been implemented as a true microservice because it is stateful, but each service instance can serve multiple independent contexts. The limits of this and the implications for scaling are to be determined later, but for load up to a few tens of contexts it seems to work fine.

The implementation is around 3000 lines of code, approximately half of which is the timeline scheduler itself and the editing operations. The rest is glue code for the web service, communication, logging, etc. This is less than 1/20th of our own SMIL playback engine.

There is one important aspect of the implementation that falls outside the architectural description of Section 4 and that has to do with synchronicity: ensuring that media items are shown at the right time relative to each other (and that that synchronicity is maintained). play and stop commands issued by the scheduler take time to reach the client components, which then need to initialize media decoder, fetch content, initialize decoders and decrypters, fill buffers, and all this on different client devices with possibly different CPU power and network bandwidth.

To address this issue we have introduced to concept of media component lifecycle. Each media item (or other user-visible component, such as menu, interaction component or – in the case of MotoGP – virtual track view or leaderboard) is handled by a media component running in a client. These components go through a fixed sequence *Idle - Inited - Started - Stopped - Destroyed - Idle*. The timeline scheduler sends commands to the client to make the transitions, and the client sends back status reports to inform the scheduler that the transition has been made.

Actually, because all clients in a context have a shared clock (based on the media time of the main broadcast video stream, synchronised between the clients using DVB-CSS) the timeline scheduler issues the transition commands in advance, with a timestamp signaling when the transition should be executed. The status reports are also timestamped, and together this has the advantage that the commands and reports can now be considered idempotent, which helps recovering from failed companion devices and companion devices joining the context at a later stage, when the show is already well underway: the layout component can simply replay the commands to the newly joined device at will (in the right temporal order).

The *inited* state and the corresponding *init* command deserve a bit of explanation. Upon receiving an *init* command the media component gets all the required parameters (such as media URL) and therefore it can start creating decoders and buffering and all that, as long as nothing is shown on the screen or audible through

[1]https://cordova.apache.org
[2]https://github.com/BBC/pydvbcss

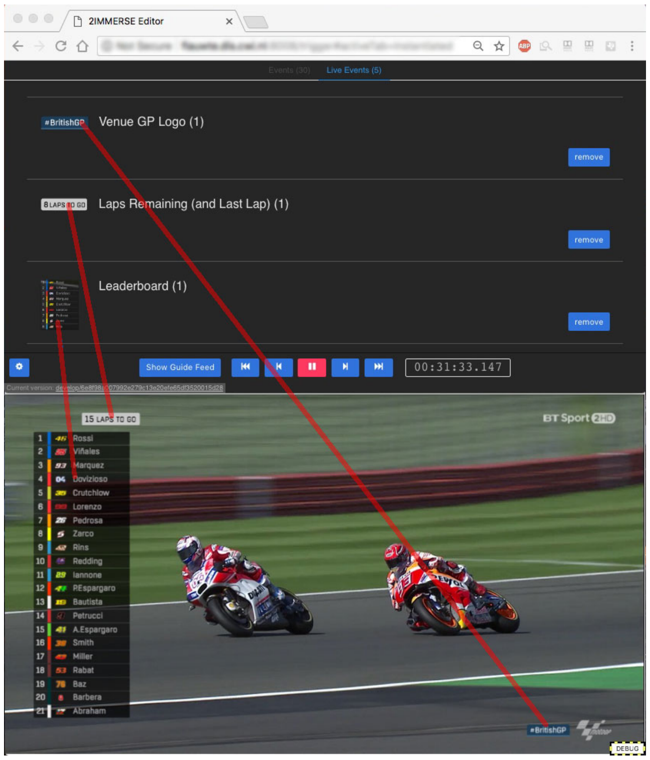


Figure 7: Production tool screen shot.

the speakers. The scheduler will emit the *init* commands as early as possible to give the media components enough time, hopefully, to be initialized when the media needs to be played.

As an example, for a `tl:par` the scheduler will emit all init calls when the `tl:par` itself gets its *init* call, and the `tl:par` will not move to *inited* state until it has received *inited* reports from all its children. On the other hand, a `tl:seq` will follow this sequence only for its first child. Then, just after the first child is started it will send an *init* for the second child. Effectively this overlaps the initialization of many media items with the playing of the preceding media item.

7 EVALUATION

The system described here has been used in a near-live setting for the MotoGP trial as part of the EU's 2-IMMERSE project. It was not a full live setting because the event, the live editing and the viewing by audiences happened in sequence over several months. This did not impact the operational emulation of a live broadcast.

The event was the British motorcycle Grand Prix race at Silverstone, August 26, 2018. All video and audio material was recorded here, and provided to the project by BT Sports and Dorna[3], the main item being the so-called "clean feed", the main video feed without any overlay graphics and such.

The production trials happened roughly in November 2018. A producer would use the production tool and preview player, watch the main video feed and trigger the appropriate media, overlay graphics and user interactions at the correct time. Figure 7 shows a screen shot of the production tool and preview player in operation. The timeline document from the production trial was saved and subsequently used in the viewer trials.

[3]https://www.dorna.com

Finally, starting in December 2018 and expected to end in February 2019, 6 sets of of 2-IMMERSE playback systems (consisting of the settop box and 2 handhelds) were sent to the homes of trialists, around 80 families in total, who viewed the race experience "as-live", so from the prerecorded material but viewed as if it was live, without ability to pause and such. The results of this are being collected and will be reported elsewhere.

The viewer-side adaptations we implemented in this trial (which are ultimately the reason for doing object-based broadcasting) fell in three areas:

- On-screen graphic overlays were shown in a size that depended on the TV screen size: on a large TV they were shown relatively smaller (so less screen space was used), on a smaller TV they were larger (so they remained readable).
- Content was adapted to the level of "MotoGP-awareness" the viewers could select. For example, the leaderboard in Figure 7 shows the "intermediate" level, the "expert" level has the names abbreviated to the 3 letter acronyms the fans will know, and the "beginner" level has not only the full names but also a picture.
- Content would be adapted to available resources. For example, the number of picture-in-picture views made available depended on available bandwidth and processing power of the playback device.

In this paper we want to concentrate on the production trial. As can be seen from the screen shot in Figure 7 we did include a pause button: it turned out that especially the more complex visual effects, such as replays which required entering of timecodes and rider names and such, were too complex to trigger live. Based on this and on other studies we will redesign the production tool to allow the workload to be shared among multiple people [9].

But from a technical performance standpoint the system performed very well. The delay between the producer pressing a button and the graphic appearing on the preview player was on the order of 0.2 second, significantly below the "few seconds" of our requirements. Timing of the graphics during the viewer trials was equally good: graphics appeared close to frame-accurate. With one exception: viewers with congested networks would incur a delay when viewing a replay. We believe this is due to idiosyncrasies of the DASH player used to play the videos and is not fundamental.

8 CONCLUSIONS AND FUTURE WORK

The development of a document format that supports the full production workflow of live broadcast television has been a longstanding interest. The delivery of quality content (at least in terms of production value!), combined with support for adaptation at the client side provides the potential for a fundamental improvement for the consumption of content in realistic modern home settings.

We have been encouraged by the initial results developed in this work. While any given implementation can always be made more robust, and any given user interface be made more intuitive, our evaluation users have reported substantial benefit in having control over partitioning content across multiple devices. In modern homes, where (multiple) secondary screens has become commonplace, the ability to have a central screen that holds the focus of a social group, while still allowing for special-purpose added-value content on personal devices, is a model with great promise. The major innovation of this paper is the development of an end-to-end control model in which a broadcast workflow can be integrated with a social client setting, without having to sacrifice production quality.

In the future, we expect to be able to expand this model to include enriched support for on-demand presentations that are not scheduled to run at fixed times. We recognize that, on the one hand, this temporal flexibility has more to do with licensing issues than content production, but we recognize the initial need to support a true-live production environment within existing workflows.

We also see great potential for the timeline document model in this paper to provide renewed support for accessible presentations, in which content versions can be adapted to the physical limitations of users. While 'motor sports for the blind' may be difficult to imagine for the sighted (or for the deaf, for that matter), we feel that existing workflows can be easily extended to expand the reach of sports content to a wider segment of society.

We conclude with the expectation that other applications of the document format presented here in traditional web/HTML applications, or in applications with multiple clocks would all benefit from a formal description and model of the timeline document format and of the suite of editing operations available. We have begun work on this formal model and hope to report on it at a future DocEng symposium.

9 ACKNOWLEDGEMENTS

The work presented in this paper was supported by the EU funded H2020 ICT project 2-IMMERSE, under contract 687655. We are indebted to our project partners in this consortium.

Our work on defining a minimalistic SMIL/NCL-like environment has been of long-term interest to ourselves and our colleagues at the Telemedia Lab of PUC-Rio in Brazil. We are greatly indebted to the efforts of our long-time friend and colleague Prof. Luiz Fernando Gomes Soares, with whom we started this work. His untimely death has had a great impact on us and on this work.

REFERENCES

[1] BBC R&D. 2016. Squeezebox. (2016). http://www.bbc.co.uk/rd/projects/squeezebox

[2] Lourdes Beloqui Yuste, Fernando Boronat, Mario Montagud, and Hugh Melvin. 2016. Understanding timelines within MPEG standards. *IEEE Communications Surveys and Tutorials* 18, 1 (2016), 368–400. https://doi.org/10.1109/COMST.2015.2488483

[3] Dick C A Bulterman and Lloyd Rutledge. 2004. *SMIL 2.0 – Interactive Multimedia for Web and Mobile Devices*. Springer. https://www.narcis.nl/publication/RecordID/oai:cwi.nl:11429

[4] Jasmine Cox, Rhianne Jones, Chris Northwood, Jonathan Tutcher, and Ben Robinson. 2017. Object-Based Production: A Personalised Interactive Cooking Application. In *Adjunct Publication of the 2017 ACM International Conference on Interactive Experiences for TV and Online Video - TVX '17 Adjunct*. ACM Press, New York, New York, USA, 79–80. https://doi.org/10.1145/3084289.3089912

[5] Romualdo Monteiro de Resende Costa, Márcio Ferreira Moreno, Rogério Ferreira Rodrigues, Luiz Fernando Gomes Soares, Romualdo Costa, Márcio Ferreira Moreno, Rogério Ferreira Rodrigues, and Luiz Fernando Gomes Soares. 2006. Live editing of hypermedia documents. In *Proceedings of the 2006 ACM symposium on Document engineering - DocEng '06*. ACM Press, New York, New York, USA, 165. https://doi.org/10.1145/1166160.1166202

[6] DVB. 2017. DVB-CSS ETSI TS 103 286-2 V1.2.1 (2017-08). (2017). https://www.dvb.org/standards/dvb{_}css

[7] Jack Jansen, Pablo Cesar, and Dick Bulterman. 2010. A model for editing operations on active temporal multimedia documents. In *Proceedings of the 10th ACM symposium on Document engineering - DocEng '10*. ACM Press, New York, New York, USA, 87. https://doi.org/10.1145/1860559.1860579

[8] Ian Kegel, James Walker, Cisco London, and Mark Lomas. 2017. 2-IMMERSE: A Platform for Orchestrated Multi-Screen Entertainment. In *Adjunct Publication of the 2017 ACM International Conference on Interactive Experiences for TV and Online Video - TVX '17 Adjunct*. ACM Press, New York, New York, USA, 71–72. https://doi.org/10.1145/3084289.3089909

[9] Jie Li, Thomas Röggla, Maxine Glancy, Jack Jansen, and Pablo Cesar. 2018. A New Production Platform for Authoring Object-based Multiscreen TV Viewing Experiences. In *ACM TVX (submitted)*.

[10] MPEG. 2015. ISO/IEC 13818-1:2015/Amd 1:2015 - Delivery of timeline for external data. (2015). https://www.iso.org/standard/67734.html

[11] MPEG. 2016. WD of ISO/IEC 23001-13 Media Orchestration (MORE) | MPEG. (2016). https://mpeg.chiariglione.org/standards/mpeg-b/media-orchestration/wd-isoiec-23001-13-media-orchestration-more

[12] Heron V. O. Silva, Rogério Ferreira Rodrigues, Luiz Fernando Gomes Soares, and Débora C. Muchaluat Saade. 2004. NCL 2.0. In *Proceedings of the 2004 ACM symposium on Document engineering - DocEng '04*. ACM Press, New York, New York, USA, 188. https://doi.org/10.1145/1030397.1030433

[13] Marian Ursu, Maureen Thomas, Ian Kegel, Doug Williams, Mika L Tuomola, Inger Lindstedt, Terence Wright, Andra Leurdijk, Vilmos Zsombori, Julia Sussner, Ulf Myrestam, and Nina Hall. 2008. Interactive TV narratives: Opportunities, progress, and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)* 4, 4 (oct 2008), 25. https://doi.org/10.1145/1412196.1412198

[14] W3C. 1998. Press Release: W3C Issues SMIL as a W3C Recommendation. (1998). https://www.w3.org/Press/1998/SMIL-REC

[15] James Walker. 2018. 2-IMMERSE: A platform for production, delivery and orchestration of Distributed Media Applications. In *IBC (submitted)*.