ActiveTimesheets: extending web-based multimedia documents with dynamic modification and reuse features

http://www.producao.usp.br/handle/BDPI/48594
ActiveTimesheets: Extending Web-based Multimedia Documents with Dynamic Modification and Reuse Features

Diogo S. Martins\(^1,2\)
\(^1\)Universidade Federal do ABC, CMCC
Santo André, Brazil
santana.martins@ufabc.edu.br

Maria da Graça C. Pimentel\(^2\)
\(^2\)Universidade de São Paulo, ICMC
São Carlos, Brazil
mgp@icmc.usp.br

ABSTRACT

Methods for authoring Web-based multimedia presentations have advanced considerably with the improvements provided by HTML5. However, authors of these multimedia presentations still lack expressive, declarative language constructs to encode synchronized multimedia scenarios. The SMIL Timesheets language is a serious contender to tackle this problem as it provides alternatives to associate a declarative timing specification to an HTML document. However, in its current form, the SMIL Timesheets language does not meet important requirements observed in Web-based multimedia applications. In order to tackle this problem, this paper presents the ActiveTimesheets engine, which extends the SMIL Timesheets language by providing dynamic client-side modifications, temporal linking and reuse of temporal constructs in fine granularity. All these contributions are demonstrated in the context of a Web-based annotation and extension tool for multimedia documents.

Categories and Subject Descriptors: H.5.1 Information Interfaces and Presentations: Multimedia Information Systems - Audio and Video

Keywords: interactive multimedia documents; web multimedia; dynamic modifications; document reuse.

1. INTRODUCTION

The adoption of multimedia technologies on the Web is highly dependent on how well user agents support them. Up until a few years ago, user agents have delegated most of the multimedia presentation functionalities to external, self-contained execution environments that could be embedded in a Web document via plugins. Tackling potential portability limitations and lack of open standards in these technologies, HTML5\(^2\) includes a number of extensions toward native multimedia functionalities on the Web, such as continuous media playback, inline vector-based graphics, state machine-based raster graphics, access to multimedia capture devices, just to name a few.

Despite these advances, HTML5 still lacks support for synchronized multimedia presentations. Even though continuous media streams can be included via specific tags, there are very limited resources to declaratively specify the temporal layout of a group of media elements. As an alternative to provide richer layouts, several authors have proposed the integration of multimedia authoring languages in the HTML5 technology stack\(^\[13, 6, 14, 18, 9, 12\]\). The SMIL Timesheets\(^1\) (or simply Timesheets) language, a derivative of SMIL\(^1\), stands out among these approaches since it includes an expressive temporal model and it provides reasonable decoupling between spatial and temporal layouts. Despite these advantages, Timesheets makes it harder to meet important requirements in Web-based multimedia authoring. In this paper, we emphasize two of such requirements: dynamic modifications and reuse features.

The first requirement, the modification of a document while it is being presented, is a recurrent design pattern in Web applications. The realization of such pattern in Timesheets documents, however, imposes some challenges. For all practical purposes, a typical multimedia document authored according to the Timesheets approach is composed of two subdocuments: the spatial document, in HTML; and the temporal document (or simply timesheet). The problem of modifying an HTML document is well understood and widely supported in user agents. Dynamically modifying a SMIL document, on the other hand, is not much understood\(^7\), and this problem also affects its derivative languages. Being XML-based, a timesheet can be manipulated via DOM operations, but the practical implications of these operations on the data structures that govern the temporal layout are not well documented in the literature. Consequently, this language still lacks a concrete method to provide dynamic modifications in an active document.

Finally, the second requirement, reuse of authoring abstractions, is advocated not only to reduce authoring costs but also as means to improve the maintainability of a resulting document. SMIL Timesheets has reuse at its foundation, since it separates temporal and spatial models. Regarding reuse on its temporal model, some features are also present, for instance the nesting of timesheets. However, important patterns cannot be accomplished with this language, such as reuse of temporal relations (for instance to refer to an internal or external temporal composition) and reuse of fragments of temporal relations (for instance to refer to a fragment of a temporal composition). The impossibility of accomplishing such use cases demands solutions to incorporate additional reuse features in this language.
In order to tackle these requirements, this paper introduces the ActiveTimesheets engine, which provides a group of language extensions and temporal formatting mechanisms to enable dynamic document modifications, granular reuse of document elements and their fragments and seamless spatio-temporal linking. The remaining of this paper is organized as follows. First, Section 2 discusses work related to the most important contributions of ActiveTimesheets. Section 3 presents preliminary definitions on the timegraph model, which is used as a basis to realize the language extensions. After that, Section 4 introduces ActiveTimesheets core extensions, namely the model of dynamic modifications (Section 4.1), the strategy for linking (Section 4.2) and reuse features (Section 4.3). Then in Section 5, implementation issues of the engine are discussed, followed by an instantiation of the engine (Section 6) in the context of a tool for enriching and extending Web-based multimedia documents. At last, Section 7 concludes this paper.

2. RELATED WORK

In order to provide richer synchronization constructs in HTML5-based documents, some authors have proposed the integration of foreign multimedia authoring languages (e.g., NCL, SMIL, or non-standardized formats) in a web document via scripting-based engines. One line of approaches (e.g., SmilingWeb [6], WebNCL [13] and NCLAWeb [14]) consist in converting, statically or dynamically, a document completely authored in the foreign language to an equivalent representation in HTML, CSS and Javascript. Such strategy makes it easier to migrate documents in the foreign language to HTML, but they make it harder to apply synchronization constructs to documents originally authored in HTML. Another line of approaches consist in augmenting HTML with timing functionality (e.g., XHTML+SMIL [18], smilText- JS [9] and Timesheets.js [4]). An advantage of this approach is that an author can take full benefit of the HTML syntax, while applying a temporal model to the document.

Regarding dynamic modifications on multimedia documents, several authoring languages tackle this problem by mapping an editing operation into the data structures that guide the spatio-temporal layout. Editing operations can originate on client-side or on server-side; additionally, the resulting update in the temporal layout can be incremental or lead to a complete recomputation of it. In certain languages, editing operations occur directly in the layout data structures: this is the case of MPEG-4 BIFS and LASeR, both of which define commands to incrementally manipulate the scene graph that is generated from a document specification. A different approach is taken by NCL, which provides both server-side and client-side live editing commands directly in the document syntax: consequently, the incremental effect of an operation is mapped, via predefined rules, into the HTG (Hypermedia Temporal Graph) that represents the document temporal layout.

In most SMIL rendering engines, the temporal layout is governed by a graph-based data structure called timegraph. The Timesheets.js API provides an experimental operation to directly edit the timegraph, in this case restricted to addition (but not removal or updates) of time container nodes. Directly editing the rendering structures is also performed by the Ambulant Annotator tool [9], but the operations are restricted to those required for captioning single videos using smilText. A more robust and author-friendly approach consists of, similarly to NCL, providing every editing operation in the authoring syntax (e.g., via DOM methods) and properly map their effects into the timegraph. A general approach to this problem is analyzed by Jansen et al. [7], which propose an abstract taxonomy of editing operations and their expected costs regarding timegraph updates. Modification operations are organized in clusters, of which timegraph updates are expected only on the Media Item (in some cases) and the Structure (in all cases) clusters. As advocated by the authors [8], this classification is a starting model to be extended by other researchers: in this paper, we take this classification as a basis and report a group of concrete methods to realize incremental updates in the timegraph.

Regarding reuse features, previous research in hypermedia authoring languages [15, 2, 16] has established patterns that allow reuse of several aspects of a document, such as: templates, internal and external references, media content, style definitions, spatial and temporal relations and compositions, just to name a few. In HTML, more granular reuse of media content is possible once the user agent supports the Media Fragments scheme [10], which provides syntax for addressing elements in the spatial, temporal, track and id dimensions. In SVG [17], intra-document reuse of geometrical shapes is possible via def and use tags, as well as inter-document reuse is accomplished via namespaces. In NCL, reuse has been thoroughly studied, which leads this language to support an ample group of reuse constructs, either internally (e.g. media content, layout, structure, etc.) or externally (e.g. nested documents, imported objects, etc.).

In SMIL, reuse of media content is possible via the src attribute of media objects and, in the case of continuous media content, media objects can be constrained to a subinterval of its intrinsic timeline via the clipEnd and clipBegin attributes. The SMIL Timesheets language augment these possibilities by introducing the timesheet element whose src attribute can refer to an external document: consequently, several timesheets can be nested in interesting reuse schemes. Additionally, the method for binding temporal specifications to spatial fragments, via CSS expressions in the item element, allows reuse of a spatial fragment in multiple item elements. Besides these improvements, some useful reuse patterns are still not possible, such as reuse of individual elements in the temporal document. In this paper, we tackle this limitation by proposing a group of techniques to reuse fragments from internal and external documents and to reuse fragments of temporal compositions (e.g. clipping of time containers).

3. THE TIMEGRAPH MODEL

The semantics of the SMIL language has been described by various formalisms, for instance, automata [1], timed petri nets [3] and logic rules [5], just to name a few. In order to compute the temporal layout of a document, the SMIL recommendation promotes a graph-based model, called timegraph. However, the SMIL literature does not include a formal definition for it, being its semantics defined mostly via natural language and sometimes pseudocode. In order to establish a basis for the extensions reported in this paper,
hereafter it is provided a working definition of the timegraph model, with a focus on timegraphs generated from documents authored in the SMIL Timesheets language.

A timegraph is a directed acyclic graph $TG = (N, R)$, where: $N$ is a set of vertices, or nodes, representing time elements; and $R$ is a set of directed edges representing temporal relationships. A node is specialized into one of various types and, being SMIL a modular language, the total set of types will depend on the language profile being used. For the present discussion, it is enough to focus on the most important groups of nodes, which are time containers (par, seq and excl), media elements (continuous and discrete) and link anchors. Timegraph relationships can be: i) **hierarchical relationships**, which represent temporal composition between time elements, for example between time containers and media elements; or ii) **event relationships**, which can represent either predictable (internal) or unpredictable (external) synchronization relationships between time elements.

![Figure 1: Example of a SMIL Timesheets document and the corresponding timegraph.](image)

Figure 1 presents a multimedia document modeling the scenario of an interactive video: when a user clicks on an overlaid menu, an alternative video is played. Figure 1a represents an external timesheet which should be included in an HTML document (omitted here) that contains the spatial layout. The presentation will initially schedule the activation of the main video (line 5), which eventually will be interrupted by the activation of an alternative video (line 7). The alternative video is scheduled by a user interaction event (sc1.click) that can occur while the menu is active (line 10).

The timegraph generated from this temporal document is illustrated in Figure 1b. Each node in the timegraph is labeled by its type and identifier. In this example, a node was created for every time container (par:1 and excl:2) and media element (either continuous media elements, e.g., cm:3, cm:4, or discrete media elements, e.g., dm:5). Notice that in this example, taking composition relationships alone, the timegraph structure is very similar to the document (timesheet) DOM, because every item element represents a single media object. In SMIL Timesheets, this correspondence does not always apply, given that an item element can represent (depending on the expression given by select attribute) multiple media objects or it can be used as a composition (i.e. similarly to a time container). Every timegraph node contains a timeline, which represents its activation interval. Given that the SMIL temporal model is hierarchical, the subgraph formed only by the nodes and the composition relationships is a hierarchy, or tree, of timelines. Every node stores its scheduling information, which is composed of the following attributes: i) the external timeline ($E$), which is the synchronization constraint between the node and its parent (via a composition relationship); ii) the internal timeline ($I$), which is the reference to enforce the synchronization constraint of the node’s children, if any; iii) the implicit duration ($di$), which can be intrinsic (e.g. for continuous media) or computed (e.g. for time containers); iv) the execution state ($s$), e.g. paused, playing, etc.; and v) the current time ($t$) of the node, which is hierarchically computed from the parent of each node.

A node can have unresolved timing because of an event relationship. In Figure 1b, event relationships are represented by a dashed line and an associated constraint, for instance the edge ($dm:5, cm:4$) in the figure. Additionally, an unresolved node have unresolved timelines, which in the figure is implied by the $Inf$ (Infinity) value in the target node of the relationship. In the example, the relationship constraint states that when the event click occurs on the element abstracted by dm:5, the time of this occurrence resolves (or is assigned to) the begin attribute of the element abstracted by cm:4.

![Figure 2: Timegraph lifecycle as a simplified state machine](image)

The lifecycle of a timegraph is abstracted as a finite state machine, whose states and most important transitions are represented in Figure 2. Part of the states represent the following timegraph processes: i) **parsing**, in which the timegraph is built from document syntax; ii) **scheduling**, which consists in computing the internal and external timelines of each node; iii) **sampling**, which consists in updating the timing of the timegraph and activating/deactivating nodes according to the schedule; iv) **resolution**, which consists in updating the timegraph schedule when some event-based constraint is solved; v) **editing**, which consists in mapping DOM modifications into the timegraph; vi) **update**, which consists in updating the schedule of the timegraph in response to resolution or editing. The last two processes, in particular, are discussed in the next section.
4. Activetimesheets

Activetimesheets is an enhanced presentation engine for the SMIL Timesheets language. Of the three modules originally included in the Timesheets recommendation, only the ones related to Timing and Synchronization are directly implemented in Activetimesheets; the remaining ones, Animation and Prefetch, are not included in the engine, as document authors can resort to analogous functionality already present in HTML5 (e.g., CSS Animations and MediaElement preload). In addition, Activetimesheets is augmented by other SMIL modules, with adaptations to reflect the particularities of the Timesheets approach. These include the Linking module, which is adapted for a seamless integration with HTML linking; and the Media Clipping module, which is extended to allow clipping of time containers.

Table 1 compares Activetimesheets with other related languages and engines. As the table demonstrates, Activetimesheets presents a set of extensions that potentially make it suitable for a greater range of use cases. Whereas it does not support the full range of SMIL 3.0 timing constructs, it extends the language in important ways. For instance, support for dynamic modifications, fine-grained reuse and media fragments are features fully provided only in Activetimesheets. Additionally, in some cases, such as media clipping and linking, the original functionality is extended to make it more conforming to novel extensions or to the HTML5 environment. The remaining of this section concentrates on the techniques that realize these extensions.

4.1 Dynamic Modifications

Editing a document while it is active has several implications in the processes of scheduling, sampling and resolution of the timegraph. From the standpoint of scheduling, manipulating document elements may require a structural or logical change in the timegraph in order to accommodate the modification. Such schedule changes also affect sampling, as some of them, for instance time manipulations, require that parameters used in the sampling process be adapted to reflect the changes. Additionally, other groups of changes may add or remove indeterminate timing attributes from the document, which also affects the occurrences of the resolution process. As a consequence, performing editing operations in the document requires some mechanism to control these processes: in Activetimesheets, this is achieved via the Timegraph Model, which also provides objects and methods to manipulate timegraph nodes. Even though this model suffices the needs of directly manipulating a timegraph, another complementary mechanism must be provided to edit the document in the authoring syntax, and more important, make sure that the document syntax and the corresponding timegraph specify equivalent scenarios.

From the standpoint of document editing, the usual mechanism for manipulating structured documents, in the Web platform, is via the DOM. Once a DOM-based editing operation is applied to a document, the rendering engine must keep the DOM and the rendering data structures in a consistent state. In Activetimesheets, a DOM inheritance strategy is adopted (Figure 3), which consists in extending the DOM by overloading a selection of methods that are guaranteed to affect the timegraph, i.e., those related to the MediaItem and Structure clusters [7]. In order that a DOM modification be properly mapped into the timegraph, procedural code in every DOM node manipulates this data structure using the

![Figure 3: Excerpt of the Activetimesheets DOM API emphasizing methods that affect the timegraph](image)
Algorithm 1

Element addition on a time container

1: function TimeContainerElement.appendChild(child)
2: super.appendChild(child)
3: if tag(child) is par|seq|excl|item|area then
4: P ← this.getTimegraphNode()
5: C ← child.getTimegraphNode()
6: P.appendChild(C)
7: C.updateAttributes()
8: C.computeSchedule()
9: P.updateSchedule(C)
10: end if
11: end function

An example of this procedure is demonstrated in Algorithm 1, which adds a new element to the document. In ActiveTimesheets, addition of new elements occurs by manipulating a composite element via its append method. First, the new element is assigned to the extended DOM node (line 2). Then, the wrapped timegraph node has its scheduling updated in order to reflect the changes (lines 3-10). In fact, the schedule is updated only if the node that was added affects the timegraph: this is the case of a restricted set of elements, depicted in the condition in line 3. Other a-temporal elements, such as meta and metatags, do not affect the timegraph and, therefore, do not require an update.

Updating the timegraph consists in a number of steps. First, the timegraph node of the DOM child is appended to the timegraph. Second, a semantic verification of the node attributes is performed, to make sure that semantic restrictions from its parent are satisfied (for instance, to disallow indeterminate begin times in the children of a seq container), opting for default values in case such verification fails. Third, the schedule of the element is computed from its container, opting for default values in case such verification fails. Finally, the timegraph schedule is updated in an incremental manner (details of this operation are discussed in Section 4.1.2). Removal and update of nodes follows a similar procedure, i.e., both the DOM and the timegraph are modified in a proper order.

Modification of attributes in the document is performed via the setAttr methods, which are overloaded by all concrete elements (refer to Figure 3), given that each element has particularities in their attributes. In general, the editing procedure consists basically of updating the attribute in the DOM and, if the attribute affects the timegraph, perform an incremental schedule update. An exception to this general procedure is the update of the select attribute in the item element, which can lead to significant structural and logical changes in the timegraph.

Algorithm 2

Attribute modification on item element

1: function TimeItemElement.setAttribute(name, value)
2: super.setAttribute(name, value)
3: if name is begin|dur|end|...|clipend then
4: T ← this.getTimegraphNode()
5: A ← this.parser.parseAttribute(name, value)
6: T.setAttribute(A)
7: T.updateSchedule()
8: else if name is select then
9: Told ← this.getTimegraphNode()
10: Tnew ← parser.parseTimeElement(this)
11: P ← Told.getAncestor()
12: P.appendChild(Tnew, Told)
13: P.removeChild(Told)
14: Tnew.updateSchedule()
15: end if
16: end function

Algorithm 2 demonstrates this procedure. The first important action taken by the algorithm is updating the DOM by assigning a new attribute (line 2). In case the attribute affects the timegraph, but it is not select, it is parsed and assigned to the associated timegraph node (lines 3-7). After that, the timegraph schedule is incrementally updated. The other situation refers to the update of the select attribute: in this case, a new sub-timegraph, corresponding to selected elements, must be generated and replaced in the timegraph. The sub-timegraph can be a single media element or a time container, depending on the number of elements retrieved

---

Table 1: Comparison between ActiveTimesheets and other SMIL-based temporal languages and engines.

<table>
<thead>
<tr>
<th>Module/features</th>
<th>SMIL</th>
<th>XHTML+SMIL</th>
<th>SMIL Timesheets</th>
<th>Timesheets.js</th>
<th>ActiveTimesheets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Timing</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>TimingAttr</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RepeatTiming</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EventTiming</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SyncBaseTiming</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SyncBehavior</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>TimeContainerAttr</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>BasicTimeCont</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>BasicExclTimeCont</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Linking</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FragmentIdentifier</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>LinkingAttributes</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>BasicLinking</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Metadata</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Novel extensions</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

+ supported - unsupported -+ partially supported ++ supported with extensions
by the selection expression. After that, the schedule is incrementally updated, starting from the new sub-timegraph.

### 4.1.2 Incremental schedule updates

An important performance requirement for a schedule update is that only the subgraph affected by the operation must be recomputed. As depicted in Figure 2, schedule updates occur in two situations: a) when an event-based constraint is solved; and b) when the document is edited. In the first case, the process consists in hierarchically propagating the resolved timing to all ancestor nodes (via traversal of the composition relationships) of every node whose timing is affected by this constraint. This is required because, due to the hierarchical temporal model, when an element has an unresolved timeline, the whole chain of ancestors have unresolved timelines as well. A full schedule computation requires traversal of all composition relationships $C (C \subseteq R)$, acting only on the subgraph formed by these relationships, the composition tree. Whereas a full schedule computation requires $O (C)$ operations, the resolution process takes $O (h)$ operations, whereas $h$ is the height of the composition tree. Differently from the resolution process, dynamic modifications do not always require propagation of updates up to the topmost ancestor. In fact, only those ancestors who have their timelines modified by the editing operation need to be updated. This process, called incremental schedule update, is demonstrated in Figure 4.

![Figure 4: Example of an incremental timegraph schedule update due to dynamic modification](image)

Figure 4: Example of an incremental timegraph schedule update due to dynamic modification

The relevant fragment of the timegraph in Figure 4a is the composition $\text{par:85}$ which aggregates an arbitrary number of discrete media elements which are executed according to $\text{par}$ semantics. Upon insertion of a new node ($\text{dm:86}$) as a child of $\text{par:85}$, the insertion procedure will attempt to incrementally update the timegraph, which will make $\text{par:85}$ attempt to update its schedule (refer to Algorithm 1). But, as the external timeline of $\text{dm:86}$ is already contained in the internal timeline of its ancestor ($([100, 120] \subseteq [0, 790])$), then no change in the schedule of $\text{par:85}$ will be observed. Consequently, no other update is expected in any further ancestor and the update procedure can stop at this point.

### Algorithm 3 Incremental schedule update

1. function $\text{TimeElement.updateSchedule()}$
2. Let $\beta$, $\varepsilon$, and $\delta$ be the $\text{begin}$, $\text{end}$ and $\text{dur}$ attributes
3. this.$d_i$ ← $\text{computeImplicitDuration()}$
4. $b$ ← $\text{computeBegin(}\beta\text{)}$
5. $d$ ← $\text{computeDuration(}\delta, \varepsilon, b\text{)}$
6. Let $i_{\text{new}}$ ← $[b, b + d]$
7. if this.$E \neq i_{\text{new}}$ then
8. this.$E$ ← $i_{\text{new}}$
9. $P$ ← this.$\text{getAncestor(}\text{)}$
10. if $P$ is defined then
11. $P$.updateSchedule( )
12. end if
13. end if
14. end function

This optimization is demonstrated in Algorithm 3, which is part of the Timegraph Model. The algorithm is an adaptation of the full schedule computation algorithm with additional verifications. The algorithm starts with computation of the external timeline of the element ($E$), from its timing attributes (lines 2-6). Then, if the computed timeline has changed, the schedule update will propagate to the immediate ancestor (lines 7-13), otherwise the update stops at this point. The example in Figure 4 presents, in fact, a best case scenario, $O(1)$, i.e., the schedule propagation stops at the immediate ancestor of the newly inserted node, involving a constant number of operations. In the case of attribute modification, the best case scenario would occur when the edited attribute do not change the element interval: in this case, the change would not even be propagated to the ancestor. Thus, it is expected that the worst case for incremental scheduling is the same as that for resolution, i.e., $O(h)$.

Another relevant optimization is the batch execution of editing operations. When multiple modifications occur in the timegraph in sequence (for instance, adding multiple children in a time container), the naive method is to call the append operation for every element, which would lead to a schedule update for every operation. In order to avoid this behavior, the ActiveTimesheets DOM API provides a number of methods for modifying multiple elements or attributes at once: for instance $\text{appendChild()}$ in $\text{TimeContainerElement}$ and $\text{setAttr()}$ in $\text{AreaLinkElement}$. The distinguishing characteristic of these methods is that they apply multiple modifications in the timegraph but call a single schedule update, in the immediate ancestor, only when all the modifications are finished, potentially leading to performance improvements.

### 4.2 Linking

The core linking features in ActiveTimesheets are based on a subset of the SMIL Linking modules, because some adaptations are necessary to sort out conflicts between constructs provided by SMIL and similar constructs provided by HTML. SMIL defines the following syntax for links: i) the $\text{a}$ element, which is similar to the homonymous HTML element; and ii) the $\text{area}$ element, which defines spatio-temporal anchors to fragments of media elements. In ActiveTimesheets, the $\text{a}$ element is dropped in favor of the HTML counterpart. The $\text{area}$ element has a temporal dimension that frames its sensitivity, which is not present in the HTML counterpart. Incorporating
**area** element is adapted to conform to the characteristics of a Timesheets-based approach.

The elaborate semantics of the **area** element allows the definition of both source anchors, in which spatio-temporal fragments can be defined, and target anchors, in which spatial fragments can be defined. In ActiveTimesheets, when **area** is used as a source anchor, spatial fragmentation is dropped, because this issue is out of the scope of the Timesheets language, whereas temporal fragmentation is supported. When used as a target anchor, ActiveTimesheets extends its semantics by allowing its application to any element with an internal timeline (e.g. including time containers and composite **item** elements) instead to only media objects.

### 4.3 Extended reuse features

ActiveTimesheets extends the reuse features in SMIL Timesheets by allowing reuse of document constructs in more granularities: *i*) named elements from external documents; *ii*) named internal elements; and *iii*) temporal fragments (clipping) of named internal or external elements. Realizing these contributions requires the development of language syntax extensions and timegraph construction patterns.

#### 4.3.1 Element reuse

Reusing individual elements of an external document has several applications in authoring. In enrichment activities, for instance, an external fragment can be reused to be annotated in a separate document. This allows a granular versioning of a document using only references to the original content. In order to explore these opportunities, ActiveTimesheets allows that any element of a document, provided that it is addressable (i.e. its id attribute is non-empty), can be reused in other contexts. This feature is demonstrated in Figure 5.

In Figure 5a, an element from an external timesheet is reused. The general scenario modeled by this document is the bookmarking of external content by reference. But, instead of bookmarking the whole document content, only a fragment of it is included. This is achieved by taking advantage of a specific addressing syntax: a fragment identifier is used in the **src** attribute of the **timesheet** element (line 3). This fragment identifier refers to a container, identified as “videos”, in the external document (omitted in the example). Additionally, the timesheet includes a series of target anchors, via **area** elements, representing bookmarks (consequently, this example is also taking advantage of linking features).

The side effect of reusing an external element is, first, the retrieval and parsing of the element fragment and, only then, the inclusion of the corresponding subgraph in the presentation timegraph. Figure 5 demonstrates this effect: the subgraphs which were included by reuse are highlighted. For all practical purposes, these subgraphs are considered as regular timegraph elements. Nodes corresponding to link anchors are prefixed with “lnk:”. The solution discussed here also allows reuse of internal elements. For that purpose, the only requirement is to use an internal fragment identifier. In summary, reuse of external and internal elements allows some compelling use cases while enriching and extending context. Additionally, the possibility of reusing temporal structures this temporal dimension in the HTML a element would require a syntactical extension to the language, which is out of scope of the ActiveTimesheets engine. In newer versions of HTML, this limitation could be minimized via inline markup using, for instance, data-* attributes.

---

### Figure 5: Example of element reuse via the src attribute of the timesheet element

![Diagram of element reuse](image)

(a) Timesheet with a reused element.

(b) Resulting timegraph emphasizing reused subgraph.

---

### 4.3.2 Element fragment reuse

One step further in reuse functionalities consists in reusing, instead of a whole element, only a fragment of it. In the video bookmarking use case, for instance, this would allow importing only a reduced clip of a group of potentially long videos. In SMIL, reuse of fragments of continuous media elements can be achieved via clipping attributes (**clipBegin** and **clipEnd**). In practice, media object clipping is a kind of non-destructive, virtual, editing operation in the temporal scope: once the media content is edited, it can be reused in different situations. The problem with clipping attributes is that they are originally applicable only to continuous media elements. In ActiveTimesheets, the semantics of these attributes are extended so that they can be applied to any element that has an internal timeline, intrinsic or not. Naturally, the usefulness of clipping temporal compositions is more appealing when an element is being reused, otherwise the same clipping result could be achieved by a proper timing scheme in the children of the composition. Figure 6 demonstrates how the timegraph is affected by clipping temporal compositions.

In Figure 6a, clipping occurs in the container **videos**, represented by node **par:1** in Figure 6b. This element is left-clipped at 15s and right-clipped at 55s (line 1), consequently its internal timeline (**T**), which unclipped would be [0, 1477], becomes [15, 55]. Thus, the element implicit duration (**di**), becomes 40s. Based on this information, the element external timeline (**E**) is scheduled to the interval [0, 40]. The main effect of clipping is that the element, when activated,
Figure 6: Example of combined clipping in item and time container

instead of starting cm:2 at 0s, will seek to 15s, being later deactivated at 55s. As the element is a composite structure, clipping this element means that all its temporal relationships are affected by the clipping. This means that the time container, when activated, will start with its internal timeline at 15s. A consequence of this is that the element cm:2 (also clipped) will never be activated, because the beginning of its active interval will occur in the past. The element cm:3 will be interrupted at 55s, before completing its original duration.

Figure 7: Timesheet reusing a clipped external element

A compelling advantage of an extended clipping model is to reuse composite abstractions: Figure 7 illustrates this syntax by adapting the example of Figure 5. This can be done for instance by including clipping attributes in a timesheet element whose src attribute refers to a named external element (lines 3-4). For instance, if a time container aggregates a video track, an audio track, a subtitle and associated discrete annotations, clipping over this container would allow reuse of a fragment of the whole composition, sliced to the specified clip. One possible use case of this functionality is, for example, the extraction of a short fragment from a long presentation.

5. IMPLEMENTATION

The ActiveTimesheets language has been implemented in a Javascript-based engine\(^5\) to be used in HTML5-compliant web browsers. The engine includes all the language extensions discussed in this paper. Figure 8 presents an overview of the most important components of this implementation.

The most important component is PresentationWrapper, which controls the whole lifecycle of the presentation, i.e., it manages the engine state machine discussed in Figure 2. This component is the main entry point of the engine, providing methods for most of the exposed functionalities. Regarding presentation setup, the wrapper, once started, commands parsing of the document and builds the DOM, using the ActiveTimesheets DOM API, and the timegraph, conforming to the Timegraph Model. The wrapper also triggers the first, full scheduling computation of the timegraph after constructing it. In addition, it uses a reference clock to control the sampling process, via a reference to the topmost timegraph node. Playback control of the presentation is also provided by the wrapper, which manages contextual conditions, such as buffering of media streams. Finally, the component exposes access methods the ActiveTimesheets DOM API so that a developer can perform dynamic modifications in the document.

6. CASE STUDY

The ActiveTimesheets engine has been integrated as a component of a Web-based enrichment and extension tool for multimedia documents. Depicted in Figure 9, the tool provides functionalities to import synchronized multimedia sessions obtained from capture environments (e.g., lectures, videoconferences, meeting recordings, etc.). An imported session is automatically transformed, at access time, in a multimedia document in HTML5 and SMIL Timesheets. Users accessing the presentation can apply textual annotations and editing commands to the multimedia document. In addition, a group of annotations can be used to generate new versions of the document containing only the fragments discriminated by the annotations. In the following discussion, it is emphasized how the extensions proposed in this paper support these activities.

6.1 Dynamic creation of annotations

A primary functionality of the tool is the creation of annotations indexed to the temporal scope of the multimedia session. From the document perspective, this involves: i) creating a media element to represent the annotation; and ii) a source anchor whose target is the interval the annotation is indexing. This is achieved using the dynamic modifications and linking features of ActiveTimesheets, consequently the user can immediately use the annotation to jump to the indexed interval, for instance.

The code excerpt in Figure 10 illustrates the use of the ActiveTimesheets DOM API to include a new annotation in the document. Annotations in the tool can have an associated editing behavior (e.g., play, pause, skip, loop, jump, etc.): in the example, an annotation with the “loop” behavior is being added to the document. First, a reference to the

\(^5\)https://github.com/diogostmartins/activetimesheets
Figure 9: A Web-based enrichment and extension tool integrating ActiveTimesheets. The multimedia document corresponds to an annotated recording of a usability experiment.

```javascript
function addLoopComment(annotation) {
  var dom = ACTIVETIMESHEETS.engine.getExternalDOM();
  var root = dom.findOne(annotation._timeline_id);
  ...;
  var node = dom.createElement('par', {id: annotation._id,
    begin: annotation.begin, end: annotation.end + 1});
  ...
  node.append(c1); node.append(c2); root.append(node);
}
```

Figure 10: Excerpt of code for dynamic modification using the ActiveTimesheets DOM API

ActiveTimesheets DOM API is obtained (line 2) and then the container of the annotation is retrieved (line 3). After this point, link anchors for the annotation are created (lines 5-7). After that, the fragment of the annotation is composed and added to the DOM using the `append` method (recall Algorithm 1). During the last `append` operation (line 8), the schedule of the presentation will be incrementally updated (as per Algorithm 3). As a consequence, the temporal and interactivity relationships defined in the annotation are immediately available in the tool.

The result of this operation is demonstrated in Figure 11. A new time container (lines 9-16) is added to the document in order to represent the new annotation. The timing of the link anchors were specified in a way that, once the first anchor (line 11) is completely traversed, the second anchor (lines 12-13) automatically seek the presentation to the start of the annotation. This is possible via event relationships and the `actuate` attribute in the second anchor. As a consequence, the presentation will keep in a loop over the annotation interval until some other event interrupts it.

**6.2 Versioning by reference**

Another important functionality of the tool is the derivation of new versions of a multimedia document based on the annotations. A user can select a group of annotations applied in the document and generate a new document containing only the fragments indexed by the selected annotations (i.e., a “summary” of the document based on annotations). With the reuse features in ActiveTimesheets, instead of duplicating fragments of the base document, the derived document simply reuses fragments of it.

Figure 11: Result of a dynamic modification.

Figure 12: Result of a versioned document generated via reuse of external element fragments

Figure 12 demonstrates the layout of a derived document. Taking as a basis the document in Figure 11, a new document was generated containing three fragments. This is done by including in the new document a `timesheet` element (e.g., line 4) with the values of the clipping attributes assigned to the interval of the base annotation. Additionally, the
Additionally, reusing elements in fine granularity potentially improves the overall efficiency and performance of the system. The ActiveTimesheets model is applicable to SMIL as well. The seq container, for example, is used to group a sequence of timesheets in a single container. This is particularly important with dynamic modifications, such as when a timesheet is bound to its host HTML document via CSS selectors: if the host document is modified (nodes are added or removed) in a way that affects the results retrieved by the selectors, then the temporal layout must be updated as well. A solution to this problem would require some change detection mechanism and an efficient expression evaluation procedure to keep the bindings updated. This also requires deeper investigation regarding the best alternative solutions and their trade-offs.

The expression power of SMIL can lead, in certain situations, to the specification of inconsistent presentations. This problem is particularly important with dynamic modifications, which can lead a document to immediately enter an inconsistent state. Given that verification of document consistency is a desirable feature in the context of dynamic modifications as well, live verification is an important theme for future improvements in ActiveTimesheets.

7. FINAL REMARKS

This paper presented the ActiveTimesheets engine, which provides important features to Web-based multimedia authoring: dynamic modifications, extended linking and reuse of elements and their fragments. These extensions have been presented considering their effects in the language syntax and in the timegraph model. Finally, a case study regarding enrichment and versioning of multimedia content have demonstrated the applicability of these extensions.

In document languages that support scripting, it is expected that procedural code can modify the host document: this is true for various Web languages, such as HTML and SVG. Consequently, realizing dynamic modifications in ActiveTimesheets brings contributions to make multimedia applications more conformant to current practices of Web development. Even though we have concentrated the discussion of dynamic modifications on SMIL Timesheets, much of the ActiveTimesheets model is applicable to SMIL as well. The core aspects of the approach, the Timegraph Model and the DOM API, can be extended to include the syntax and semantics of other elements and attributes of the SMIL language. From the standpoint of reuse features, referencing internal or external elements has several applications in authoring multimedia documents. On one hand, reuse enables non-destructive extension of external content, as enrichments can be applied in a decoupled manner over a document fragment. Additionally, reusing elements in fine granularity potentially reduces the verbosity of the document.

An important issue that is not tackled by ActiveTimesheets is that of spatio-temporal binding consistency. This problem consists in keeping the temporal layout updated if a change in the spatial document occurs. This is particularly important because a timesheet is bound to its host HTML document via CSS selectors: if the host document is modified (nodes are added or removed) in a way that affects the results retrieved by the selectors, then the temporal layout must be updated as well. A solution to this problem would require some change detection mechanism and an efficient expression evaluation procedure to keep the bindings updated. This also requires deeper investigation regarding the best alternative solutions and their trade-offs.

The expression power of SMIL can lead, in certain situations, to the specification of inconsistent presentations. This problem is particularly important with dynamic modifications, which can lead a document to immediately enter an inconsistent state. Given that verification of document consistency is a desirable feature in the context of dynamic modifications as well, live verification is an important theme for future improvements in ActiveTimesheets.

Acknowledgments. This work was supported by funding from CNPq (process no. 143144/2009-0) and FAPESP (process no. 2013/03337-6). We would like to thank Jack Jansen, Pablo Cesar and Dick Bulterman, for valuable insights of the SMIL language while the first author was visiting CWI, Amsterdam, Netherlands.

8. REFERENCES