Media Abstract Sequencing Template (MAST) Format Specification

Contents

Revision history 1

Introduction 2

Purpose 2

MAST format 2

Format Summary 2

Mast XML element summary 3

Mainsail – MAST Engine reference implementation 6

Mainsail Summary 6

Mainsail Design 6

Future 8

IMastAdapter 8

Appendix A: MAST document sample 9

Revision history

Version	Notes	Authors	Release Date	Reviewed by
0.9	Original draft	Nick Brookins, James Mutton, Will Law, Charles Newman	03/30/2009	Nick Brookins, James Mutton, Will Law, Charles Newman, Jamie Sherry

Introduction

The Media Abstract Sequencing Template (MAST) is a declarative language and markup specification based on XML that defines a set of triggers that define when events called triggers should be acted upon based on a set of associated conditions. MAST is currently designed in conjunction with efforts in the Open Video Initiative to be an open solution to address sequencing and layout for advertising at the player level. While MAST has the potential to satisfy a wide range of use cases beyond advertising, they are beyond the current scope.

Purpose

MAST is proposed by Akamai as an open standard for the sequencing of in-stream advertising content. MAST is designed to be compatible with and supplemental to other standards such as VAST and VPAID which have been designed by the Digital Video Committee of the Interactive Advertising Bureau (IAB). While VAST defines the content of the advertising experience and VPAID provides a standard for ad-player interaction, there is also a need for a common approach to address sequencing and resolve the issue of having different sequencing implementations for every deployment. Our intent is for MAST to become an IAB recommended format for use in sequencing advertising within a media player and further the value of having a standards-based approach for in-stream advertising.

MAST format

Format Summary

MAST is a declarative language and markup specification based on XML. It defines the 'where' and 'when' with events and triggers that cause player activity based on a set of conditions. It references the ad content via a link to a VAST document, the payload of the trigger.

The basic structure of the MAST document is:

- Triggers within the mast document
- Conditions defining when the triggers are fired, or ended
- A payload providing information to the process consuming the trigger

The XML Schema Definition (XSD) of MAST for developers can be found in here: http://openvideoplayer.sf.net/mast/mast.xsd. A sample MAST document can be found in Appendix B.

MAST is acted on by a sequencing engine in the player, which can be implemented in Silverlight, Flash, or even JavaScript. The MAST document can be provided to the player in a playlist or web page, the latter of which consists of using the object tag parameters or JavaScript. See the Mainsail section for more information regarding a MAST Engine implementation.

Mast XML element summary

Root node

```
<MAST>
```

This is the root node.

- Attributes:
  - version - the version of the MAST spec that the document represents
- Children:
  - triggers - the container element for the list of MAST Triggers

triggers element

```
<triggers>
```

This element is a container for the individual *trigger* elements.

- Parent: MAST
- Children: trigger (multiple)

trigger Element

```
<trigger>
```

This element describes an individual trigger.

- Parent: triggers
- Attributes
  - id - (string) a reference for this trigger
  - description - (string) user provided description, for reference purposes only
- Children
  - startConditions
  - endConditions
  - sources

startConditions Element

```
<startConditions>
```

This element is a container for the individual *condition* elements that affect the initiation of the trigger.

- Parent: trigger
- Children: condition (multiple)

endConditions Element

`<startConditions>`

This element is a container for the individual *condition* elements that affect the revocation of a trigger.

- Parent: trigger
- Children: condition (multiple)

condition Element

`<condition>`

This element describes one condition of this trigger, when a condition evaluates true it can initiate a trigger or revoke it, depending on its parent (startConditions vs endConditions). Multiple condition elements are treated as an implicit OR, any one of them evaluating true will fire the trigger. Child conditions are treated as an implicit AND, all children of a condition must evaluate true before a trigger will fire (or be revoked) from that condition. Note that event-type conditions cannot be child conditions - only property conditions can.

- Attributes
  - type - (enum) specifies the type of condition, currently 'event' or 'property'
  - name - (string) the name of the property or event to be used in evaluation
  - value - (string) the value that a property will be evaluated against
  - operator - (enum) the operator to use during evaluation: EQ, NEQ, GTR, GEQ, LT, LEQ, MOD

- Parent: startConditions, endConditions, condition
- Children:
  - condition (multiple)- a sequence of child conditions which will must also evaluate for this condition to fire the trigger

sources Element

`<sources>`

This element is a container for the individual *source* elements that define the payload for the trigger. Each triger can have multiple sources; any sources at the top level are handled independently. Each source can also have a *sources* element which can contain child sources. These children are only handled if the parent source succeeds. This can be useful behavior in the case of situations like companion ads which are tied to a linear ad.

- Parent: trigger, source
- Children: source (multiple)

source Element

`<source>`

The source element describes a payload for the trigger, which will be acted upon when the trigger fires.

- **Attributes**
  - **uri** - (string) the uri to retrieve a source from, for example a link to a VAST doc.
  - **altReference** - (string) This is used to key the source against a resource already known by the player. To prevent collisions it can be keyed with the uri when possible.
  - **format** - (string) the format this source is in, to be used to determine a handler for the payload. Examples would be 'vast', 'uif', etc.

- **Parent** : sources
- **Children**: targets, sources

targets Element

`<targets>`

This element is a container for the individual *target* elements. Multiple targets are handled independently.

- **Parent** : source
- **Children** : target (multiple)

target Element

`<target>`

This element describes the target for the payload, the player will use this for placement when the trigger is fired. A target can be the child of another target, which implies a dependency - if the parent target fails to place correctly, the child will be skipped.

- **Attributes**
  - id - (string) the type of target, can be used for keying particular payload items to a target
  - regionName - (string) a named region / container or other target that can be used by the player
- **Parent**: targets, target
- **Children**: target (multiple)

Mainsail – MAST Engine reference implementation

Mainsail Summary

Mainsail is an engine that handles a MAST document. While MAST itself defines the format of the XML document, the engine does the work of evaluating triggers and informing the owner/player when triggers are ready to activate or deactivate, passing the payload along for use by the appropriate handler. Mainsail is the reference implementation for MAST and has been completed in C# for use in the Silverlight OpenVideoPlayer. An ActionScript version is also under development for the Flash OVP.

While Mainsail is used as an example of handling the MAST format, this could be accomplished in other ways. For example a player or other application could have it's own sequencing logic that used MAST as an input format. A MAST engine could support just a subset of MAST features for specific applications. A MAST engine could be implemented in nearly any programming language, including JavaScript.

Mainsail Design

This section will discuss the Mainsail implementation in C#. Mainsail for C#/OVP is a distinct plug-in, with a project called MASTSequencing which compiles to MASTSequencing.xap. This plugin will be available in the next release of OVP/Silverlight, expected to be OVP version 2.2.000.

MAST Input / Parsing

At the highest level there is a class called Mainsail, which is what other application components would interact with. The engine also has a MAST parser, which is auto-generated from the XSD schema using an free tool called XSD2Code. This tool creates classes for each major element in MAST, including Trigger, Condition, Source, and Target. Each class has a deserialize method, typically the parsing is done by passing a full mast document xml string into the deserialize method of the top-level MAST class, generated by XSD2Code. The Mainsail class has methods to load MAST from a string of XML, a URL to the document, or by sending an instantiated Trigger' object - these methods use the parsing classes where applicable.

Initialization

When MAST documents are added to Mainsail, it loops through each trigger. For each one a TriggerManager class is instantiated, and passed a reference to the Trigger it will be handling. The logic for a trigger was put into the Manager class so future schema changes could regenerate the Trigger and other classes without loosing the associated logic. Once a TriggerManager is created, Mainsail attaches to its Activate and Deactivate events.

Inside TriggerManager in creates a ConditionManager for each of it's start and end conditions and attaches to the Condition's "EventFired" event, this will be explained further in a later section.

Condition Property Evaluation

Mainsail has a timer with a configurable period, typically about 250-500 ms. On each timer loop Mainsail calls the Evaluate method on each TriggerManager, which causes them to evaluate their conditions. Property-type conditions are evaluated directly: the TriggerManager calls an evaluate method on each condition. The condition uses the property name attribute to obtain the current value of that property, for example 'Position' might return 10.5 seconds.

The property value is returned from the IMastAdapter - which defines the properties and events that the engine knows how to evaluate against. This interface should be implemented by the owner of the engine, or be implemented by another class that can get the appropriate properties and events on behalf of the engine. In the case of the C# Mainsail, IMastAdapter is implemented by OvpMastAdapter, which translates the properties and events to those exposed by OVP. In other applications these properties and events may not be relevant, and a totally different interface may be used.

The value retrieved from the property is compared against the value attribute of the condition, using the specified operator. For example, if the operator was GTR (Greater or Equal) and the value was 8 sec, then this condition would evaluate true. If true, the condition also evaluates its own child conditions, which are treated as a logical 'AND'. If all of them are also true, then the ConditionManager returns true from the evaluate call - letting the TriggerManager know it is time to fire the 'Activated' event. If one condition fails evaluation, the TriggerManager moves to the next, as peer conditions are treated as logical 'OR'.

Event Conditions

Event Conditions are handled a little differently. The condition has the name of the event is is tied to, and it looks for this event on the IMastAdapter. If found it dynamically attaches to it. Now if the object that is implementing IMastAdapter fires this event, it is handled by the Condition. The condition first evaluates any child conditions, which would also have to evaluate properly for this trigger to activate. If children also evaluate, the condition fires an event that bubbles up to the TriggerManager, and finally to Mainsail itself. Because events are asynchronous, polling is not required.

Activation

As discussed, a TriggerManager fires 'Activated' when its property conditions evaluate properly, or when an event fires through a condition. Mainsail handles this, and calls TriggerActivated, a method exposed by IMastAdapter. Is passes a reference to the trigger, which can now be used by the consumer to handle the payload appropriately. At this point Mainsail relinquishes control, as the payload is implementation specific and should be handled by the consuming application.

In the case of the C# OVP Mainsail, the OvpMastAdapter receives the activated trigger. It loops through each source on the trigger and looks for a player plugin that can handle the format specified by the trigger.source format attribute. When it finds an appropriate plugin, it gives the payload to this handler. In the current example this is likely a VAST document, which would be handled by the VastAdHandler class/plugin.

Deactivation

When a payload has completed, for example a linear ad reaches its end, the Handler will call back to the IMastAdapter DeactivateTrigger method. This will clean up any references and also inform Mainsail that the trigger has completed. During the lifetime of a trigger, Mainsail evaluates the end conditions of the Triggers in the same way that it previously did for start conditions. If an end condition evaluates true during the life of a trigger, Mainsail will directly call DeactivateTrigger on IMastAdapter - revoking the payload from the other direction.

Future

At this point the MAST spec has stabilized and no major changes are expected soon. Future changes to Mainsail could include updates to the IMastAdapter interface, to support additional properties and events. We may want to consider a naming convention, so that multiple interfaces could be defined, each with the relevant members for various applications. There will also be new implementations of Mainsail, such as in ActionScript/Flash for OVP.

In C# Mainsail itself, no changes are expected as new payloads are introduced. For example, if we implement a handler for a new Ad tag format, like Eyewonder's UIF, it would involve creating a new handler for this payload format, which would be transparent to Mainsail. These handlers can be in a different plugin from MAST/Mainsail, or in the player/application itself.

IMastAdapter

IMastAdapter is an example of the interface that Mainsail, or another MAST Engine, uses to query properties and attach to events for condition evaluation. This process is discussed in the Mainsail section. As MAST could be used in many different applications, there may be a need for different interfaces.

This interface is designed for use in a Media Player application, and has all properties and events that a condition can reference. The code is available at http://openvideoplayer.sf.net/mast/IMastAdapter.cs

Appendix A: MAST document sample

The following is a sample that demonstrates a pre-roll with companion banner

```
<MAST xsi:schemaLocation="http://openvideoplayer.sf.net/mast
http://openvideoplayer.sf.net/mast/mast.xsd">
  <triggers>
    <trigger id="preroll" description="preroll before every item">
      <startConditions>
        <condition type="event" name="OnItemStart">
           <condition type="property" name="Duration" operator="GEQ" value="1:00"/>
           <!-- This child condition must also be true, so the pre roll only triggers
                with content 1min or longer -->
        </condition>
      </startConditions>
      <endConditions>
        <condition type="event" name="OnItemEnd"/>
        <!-- This 'resets' the trigger for the next clip-->
      </endConditions>
      <sources>
        <source uri="http://api.atdmt.com/sf=VAST_PreRoll_XML_V2;" format="vast">
          <sources/><!--Child sources, which would be dependant on this one -->
          <targets>
            <target region="Banner1" type="banner"/>
            <target region="VideoArea" type="linear"/>
            <!-- target can be assumed for linear, but is explicitly defined here-->
          </targets>
        </source>
      </sources>
    </trigger>
  </triggers>
</MAST>
```