Cinelab model Documentation

Release 1.1

LIRIS

July 26, 2012

CONTENTS

CHAPTER

ONE

CONTEXT

Advene (Annotate Digital Video, Exchange on the NEt) is an ongoing project in the LIRIS laboratory (UMR 5205 CNRS) at University Claude Bernard Lyon 1. It aims at providing a model and a format to share annotations about digital video documents (movies, courses, conferences...), as well as tools to edit and visualize the hypervideos generated from both the annotations and the audiovisual documents. Teachers, moviegoers, etc. can use them to exchange multimedia comments and analyses about video documents.

The data model supporting the Advene application offers, in addition to the annotation storage itself, an explicitation of their structure (through notions of schema and annotation type), as well as an explicit view concept (templates applied on data to produce hypervideos) and queries. The following figure sums up the principle of the use of hypervideos.

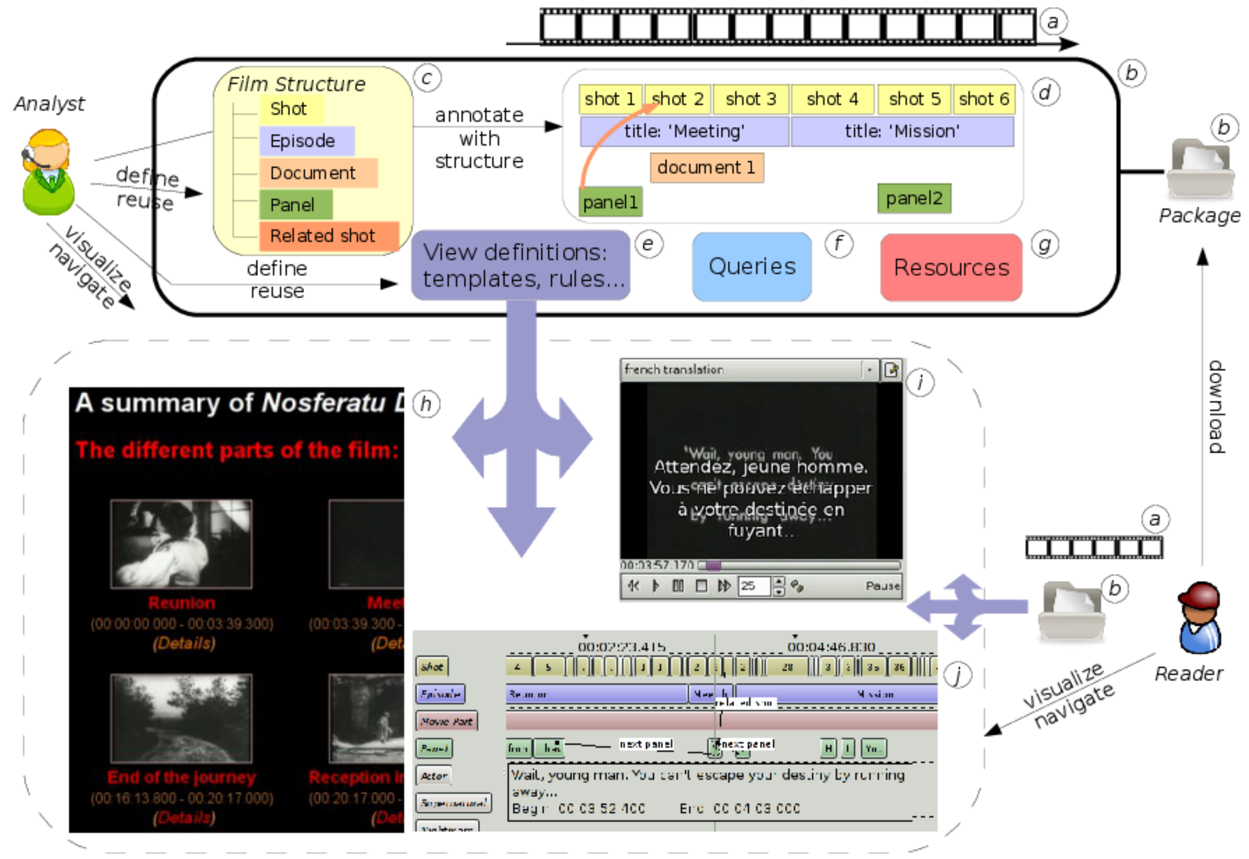

Figure 1.1: Hypervideo creation and use in Advene

Collaborations with other partners like IRI during the Cinelab project (2007-2008) led us to identify weaknesses in the first version of the data model, and to propose an evolution addressing these issues. This document specifies the new model and format, used for exchanging audiovisual metadata among Cinelab project partners, and more widely as a storage and exchange format for various applications (especially Advene and Lignes de temps). The Cinecast projet (2009-2012) allowed us to confront these evolutions to new application fields and other partners, and validate their relevance.

CHAPTER

TWO

INTRODUCTION

This document is composed of 4 parts.

First, a presentation targeted at users of concepts and vocabulary used in the project. It gives a big picture of the different elements.

Then, we define the hypervideo model, designed on the basis of our work in the Advene project. It defines a minimal data structure needed to implement hypervideos.

The next part presents the Cinelab Applicative Model and more precisely specifies and brings operational constraints on the abstract model presented before, in order to get an implementation of the notions presented in the concepts and vocabulary part.

Finally, the last part specifies an XML+ZIP and a JSON-based serialisation formats for Cinelab data, so that they can be shared among applications.

CHAPTER

THREE

CINELAB GLOSSARY

This part aims at giving a user-targeted description of the concepts used in Cinelab.

3.1 Package

A package is a consistent set of references to medias, description schemas, annotations, relations, views, resources, queries, sets (in the form of list or tags), and other packages (dynamically imported/referenced).

Elements from a dynamically imported package can be accessed from the importer package.

A dynamic import is a simple *reference* to the imported package. The content of the imported package is available "in" the importer package, but cannot be directly modified from the importer package. If data has to be modified, then the user has to import it by *copying* the original elements into the desired package. Each copied element would then be duplicated, and both instances (original and copy) could be modified independently.

3.2 Annotation

An annotation is data - called *annotation content* - linked to a temporal fragment of a movie.

An annotation belongs to a type (a category).

3.3 Annotation type

An annotation type offers a way to categorise annotations. This categorisation has two goals:

1. offer an **intrinsic and native** way of grouping a set of annotations, identified by a name/id *(the annotation is created as an instance of this type)*
2. specify some of the characteristics of its annotations, typically constraints over its content (for instance: free text, image, structured contents with constrained fields, etc) *(the annotation content is constrained)*

A "General annotation" type always exists.

3.4 Relation between annotations

A relation can link multiple annotations - called relation members - and can possess a content. It belongs to a type (a category).

For instance, a binary relation between two annotations of type *Shot* may allow to describe a narrative consequence between two shots, describing this consequence in the relation content.

3.5 Relation type

A relation type categorises relations. This categorisation has two goals:

1. offer an **intrinsic and native** way of grouping a set of relations, identified by a name/id *(the relation is created as an instance of this type)*
2. specify some of the characteristics of its relations, typically constraints over the number and type of their members (for instance: link exactly 2 *Shot* annotations) and their content (for instance: free text, image, structured contents with constrained fields, etc) *(the annotation content and properties are constrained)*.

A “General relation” type always exists.

3.6 Resource

A resource is a data file linked to a package. It can be involved in view renditions.

3.7 Group

A group allows to define element categories in a more flexible way than the annotation types and relations types :

- the same element can belong to many groups (while an annotation has one and only one annotation type).
- a group can contain heterogeneous elements (annotations of different types, annotations and relations, etc.)

Any element (annotation, relation, schema, type, view...) can be associated to a group.

Groups are expressed as lists (enumerations) and *tags* in the model.

3.8 Query

A query returns a set of elements from the package that match a number of criteria.

3.9 Description schema

A *description schema* is a way to describe the structure of a movie annotations.

A description schema defines categories of annotations (possibly specifying their contents), ways of linking annotations (through typed relations), etc., that it to say annotation types and relation types.

3.10 View

A view is a way to present a set of annotations and the videos they are linked with. As is often the case, there can sometimes be a confusion between the *view specification* (for instance a HTML template with processing directives) and the *view rendition* (with the previous example, the generated HTML document, where directives inserted data from specified elements). Thus, a view can be transmitted by transmitting its specification (then the directives processing will be carried out by the destination), or by transmitting its rendition. The precise term (specification or rendition) will be used in case of ambiguity.

For instance, a HTML document presenting a table of contents for a video, generated from annotation indicating different sequences and offering a direct access to the video, is an example of view that can be qualified as *static*. The specification of this view can be realised through a *template* language, as is done in Advene.

Overlaying the video with textual information (*captioning*) is another view example, that can be qualified as *dynamic*. The specification of this kind of views needs specific formats and models, such as a Event-Condition_Action as done in Advene, or a declarative specification as done in SMIL.

CHAPTER

FOUR

HYPERVIDEO MODEL

We propose in this section a conceptual model for hypervideos, so that they can be created, stored and shared.

Our model is composed of two-layers. The first layer, named *core model*, aims at being general enough to match a number of uses, as independently as possible from technological evolutions. The second layer, named *applicative model*, specialises the core model through a number of technical decisions that make it directly implementable. Multiple applicative models can be proposed over the core model, but the Cinelab project aims at finding a common applicative model that can be useful for many partners/uses.

This section focuses on the core model, indicating the points that need to be more precisely specified in the applicative model.

4.1 General points

4.1.1 Package

The *package* is the documentary unit of the hypervideo model. It contains a set of elements, linked through different relations (see below).

A package can be identified by a URI, that can identify it persistently (independently from the way it was obtained). When a package does not have a URI, it is identify through the URL used to access it.

4.1.2 Elements

All elements in a package are uniquely *identified* by a character string composed of alphanumerical characters, dashes, underscores and colons (:), matching the following regular expression:

```
^([a-zA-Z_][a-zA-Z0-9_\-]*|:[a-zA-Z0-9_:\-]*)$
```

NB: if a colon (`:`) is used in an identifier, then the identifier **MUST** begin with `:`. This constraint is necessary for the correct handling of dynamic imports (see section Dynamic import).

Each element belongs to an element type, among those defined in section Elements. Two elements, even of different types, cannot have the same id in the same package.

If an applicative model wants to define a specific (internal) role for an element, it should use for this element an id starting with `:`. Applications should not allow users to use ids starting with `:` other than those defined by applicative models.

An element id can be used as a fragment on the package URI, in order to identify the element from out of the package. Given a package whose URI is `http://advene.org/packages/example1.czp`, containing an element with id `a1`. This element can be addressed (outside of the package) with the following URI: `http://advene.org/packages/example1.azp#a1`

4.1.3 Metadata

The package and any of its elements can be enriched with any metadata, defined or not in the model. This metadata is made of (key, value) couples, where key is an arbitrary string (it SHOULD be an URI, to conform to the RDF metadata model); value can be an arbitrary string or a reference to an element.

Applicative models can require or suggest to use some metadata for packages, elements in general of element of a specific type. In this case, they will specify the appropriate keys, and possibly the authorized values for the keys.

4.1.4 Content

In the following text, we call *content* an octet string with a MIME type. The MIME type describes how the octet string should be interpreted.

A content can also optionaly be declared as conforming to a model (for instance, XML schema, Relax NG, JSON-schema, etc). The model itself should in this case be store in a *package resource* (see below) and the content will reference this resource. The validation of the content with respect to its model depends on the model MIME type. A list of valid MIME types for models (for instance `application/relax-ng-compact-syntax` or `application/schema+json`) should be specified by applicative models.

4.2 Element type

This section describes the various element types that can be part of a package. There are roughly three kinds of types: those related to the annotation structure (media, annotation, relation), those related to the package structure (lists, tags, queries, dynamic imports) and those related to annotation presentation (views, resources). Some element types can provide common *interfaces*. These abstrac interfaces (group, pipeline) are also presented.

4.2.1 Group(interface)

An element matches the *group* interface if it defines a subset of its package's elements. A group allows to enumerate all its elements, as well as its elements matching a given type. Applicative models can define a specific order with which some element types instances should be enumerated (for instance, chronological order for annotations).

4.2.2 Pipeline (interface)

An element matches the *pipeline* interface if it takes an element or a package as input, and outputs an element or a package. The output element can be an pre-existing package element, or a generated element.

Applicative models can specify an *integrity constraint* mechanism indicating on which elements a pipeline can apply. This mechanism can use *Test* views as defined below.

4.2.3 Media

A package references a number of audiovisual medias. Each of this medias is represented by a Media element, which features a URL addressing the corresponding audiovisual media. The applicative model can specify usable URL schemes (especially for accessing media without standard URLs such as DVDs). A Media element also specifies

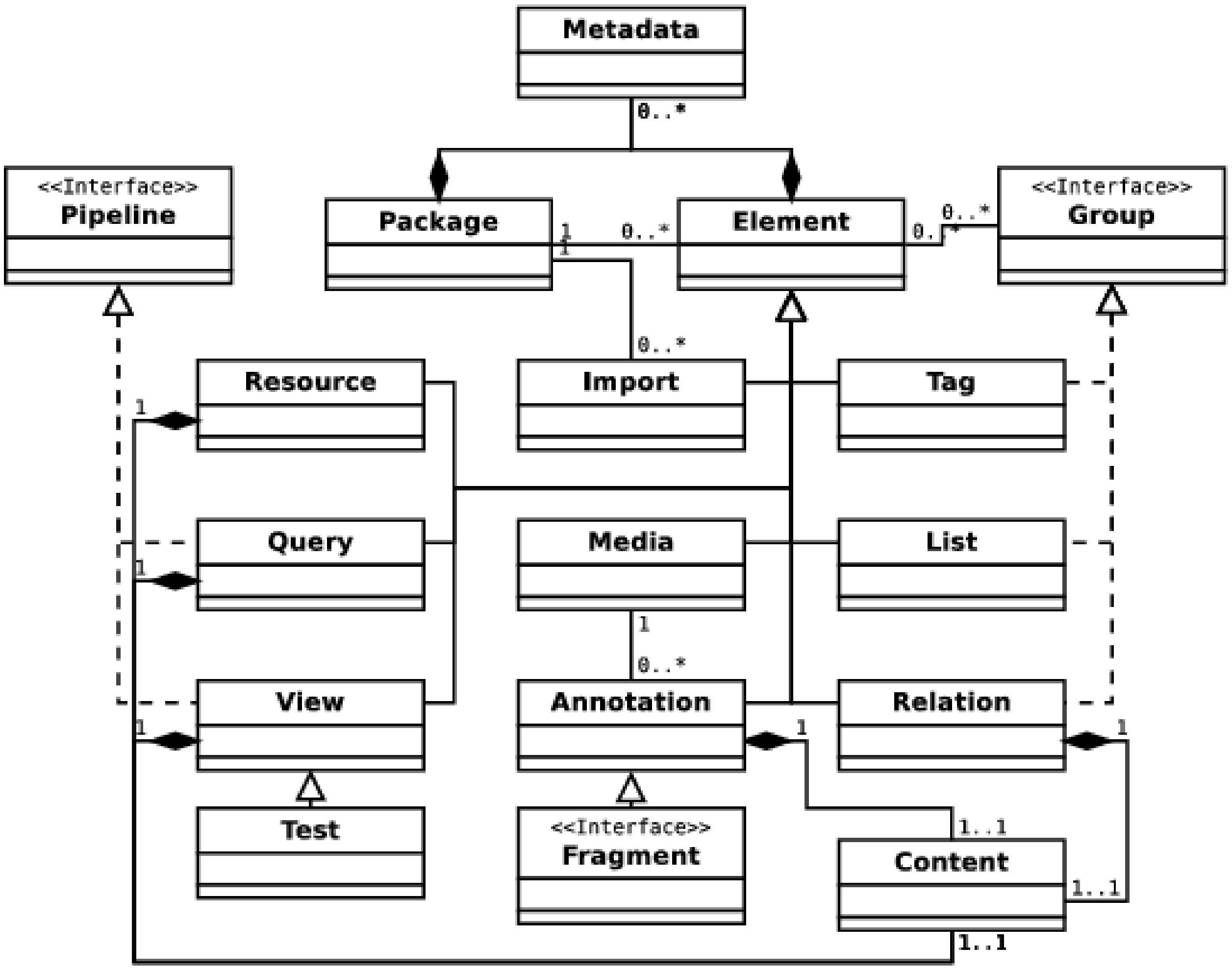

Figure 4.1: Overview of the interrelations of element types.

a temporal frame of reference, specifying a unit and an origin, used to address media fragments. For instance, the `m1` media uses milliseconds units and starts at 123ms. Applicative models MUST specify valid frame of reference parameters.

Once a package is shared, it is possible that a media URL specified in the package is not accessible by the recipient: link to a local file, DVD, etc. It is recommended that Media elements are enriched with metadata allowing other users to identify the media so that they can either localize it an/or check that an available media is "compatible" with the one specified by the author (for instance: title, duration, ISAN number, etc). The core model does not specify this metadata (which depends on the technological possibilities). Applicative models are strongly encouraged to specify such metadata.

4.2.4 Annotation

An annotation is composed of a content, linked to an audiovisual media fragment. The fragment is made essentially of three elements: the id of the annotated media, a start timecode and an end timecode. Temporal bounds are expressed by integers, expressing units specified by the Media element.

Some applications may need audiovisual fragments more complex than a simple temporal interval: spatial-temporal interval, MPEG-4 object, audio track specification (for a DVD), etc. For these scenarios, the corresponding applicative model will specify metadata to augment the annotation with, in order to constrain the annotated fragment with appropriate information.

A specificity of the model is that the fragment element is not separate from the annotation: each annotation intrisically defines the media and temporal interval its is linked to. Other approaches commonly define fragment elements independently from the annotations, so that a single fragment instance can be linked to multiple annotations. In our model, annotations are independent (time-speaking) one from another: a fragment cannot exist by itself, since simply defining a fragment is implicitly taking position on some semantics for the fragment, thus annotating it. These semantics are most often expressed through an annotation content. From this point of view, it is impossible to discriminate wether having the same timecodes for 2 annotations means that they are linked or not. It depends on the semantics of the annotations, thus the model must remain agnostic about it.

4.2.5 Relation

A relation defines an *ordered* set of annotations. The annotations are the *members* of the relation. A relation MAY also have a content.

A relation implements the Group interface for accessing its members.

4.2.6 View

A view can produce a rendition for a package or a package element, possibly using elements issued from associated audiovisual medias and resources.

Views implement the Pipeline interface and always output a resource (see below). Views possess a content (their definition), whose MIME type determines how the view is interpreted. The list of valid MIME types for view definitions must be defined by applicative models.

Test

Some views may produce a content interpretable as a boolean value. Such views can then be used to discriminate elements in a set, they are called *Tests*.

4.2.7 Resource

A *resource* is composed of a content (data + MIME type). It does not reference a specific audiovisual media, so does not strictly belong to the annotation structure. It can be useful to build some views.

4.2.8 Dynamic import

Elements defined within a package are called the package's *own elements*. It is also possible to reference from within a package elements defined by another package. These are called *dynamically imported elements* (or, when there is no ambiguity, *imported elements*).

A *dynamic import* is an element referencing, through its URL, another package. The applicative model can specify the usable URL types. The *dynamic import* element also stores, when possible, the imported package URI.

It is possible that, when a package is shared, the defined imported package URLs are not available for the recipient: unshared local file, unavailable/unaccessible server, etc. It is recommended that *Dynamic import* elements are enriched with metadata allowing other users to identify the package so that they can either download or request it an/or check that an available package is "compatible" with the one specified by the author (the package URI is of course the first way to check a package identity, provided it includes versioning information). The core model does not specify this metadata. Applicative models are strongly encouraged to specify such metadata.

It is not required that the dynamic import structure is acyclic. It is possible that two packages import each other. However, a package MUST NOT have two dynamic imports to the same package, and cannot import itself.

Constraint: the id of a *dynamic import* element MUST NOT contain the colon `:` character, so that its elements can be correctly addressed (see below).

Ids for dynamically imported elements

When a package dynamically imports another one, we need a way to identify the imported package's elements from within the importer package. Using URIs with fragments is possible, but not always convenient. We then propose to use the notion of *identifier reference* (id-ref).

Given a package *p1* defining a dynamic import *i*. *i* references a *p2* package, containing an element with the *e* (in the context of *p2*). The *e* element can be identified within *p2* by concatenating the identifier of *i*, the colon `:` character, and the identifier of *e* (in the contexte of *p2*). The obtained identifier is the *identifier reference* (id-ref) of *e* within *p1*.

Example: if *p1* imports *p2* through an import identified with `foo`, and *p2* contains an *a1* annotation, then the id-ref of the annotation in *p1* is `foo:a1`. If *p2* contains an annotation `:toto:a2` (using a `:toto` internal prefix), it id-ref in *p1* is `foo::toto:a2`.

This schema can be used for multiple import levels. For instance, let us assume that *p2* imports *p3* with the id `bar`, and that *p3* contains a `a3` annotation (id-ref in *p3*). Its id-ref in *p2* is then `bar:a3`, and its id-ref in *p1* is `foo:bar:a3`.

NB1: The prohibition on using `:` in dynamic imports identifiers ensures that an id-ref can be unambigously interpreted.

NB2: When the distinction between id and id-ref is not relevant, we simply use the word "identifier".

Direct and indirect imports

Examples presented below allow to discriminate between *directly* imported element (i.e. own elements of a package imported by the current package, for instance `foo:a1`) and *indirectly* imported elements (i.e. elements imported by a package, itself imported by the current package, for instance `foo:bar:a1`).

A constraint is that only directly imported element can be referenced by a package's own elements. For instance, if the *p1* package from the previous example defines a *r1* relation, any own annotation of *p1* can be a member, as well as the own annotations of *p2* (directly imported in *p1*, through the `foo` dynamic import). In contrary, the `foo:bar:a3` annotation, indirectly imported by *p1*, cannot be a member of the *r1* relation. To solve this issue, one has to create a direct import of *p3* in *p1*. Concerned references include: the media associated to an annotation, a content model, members of a relation, list items, tag-element associations, metadata values.

Ergonomically, this limitation is reasonable and technically, it has good properties: a directly imported element has a unique id-ref (as opposed to indirectly imported components, which can have several). By limiting the number of intermediaries, the risk of link breakage is limited. Finally, we can find the URI of a directly imported element with the available data of the package defining the dynamic import, while it is impossible for an indirectly imported one.

4.2.9 Query

A *query* implements the Pipeline interface, and always produces a list (see below). If a query produces items external to the package (for instance, URLs), it has to encapsulate them in resources (temporary, but that can be saved), so that is indeed produces a list of package elements.

A query has a content. The MIME type of the content determines how the query is interpreted. Le list of valid MIME types for queries must be specified by applicative models.

4.2.10 List

A list is a sequence, defined by extension and ordered by the user, of package elements (own or directly imported). Lists implement the Group interface.

Applicative models can specify a *contraint integrity* mechanism, indicating which objects can belong to a list. This mechanism can use Test views as defined above.

4.2.11 Tag

A Tag is an element that can be *associated* with any other package element (own or imported). It allows to group a number of elements that share a characteristic.

Applicative models can specify a *contraint integrity* mechanism, indicating which objects can be associated with a tag. This mechanism can use Test views as defined above.

Tags and dynamic imports

Tags differ from lists in that they do not define the order of their elements, and also by the fact that they are "open", which means that a package can dynamically import a tag an associate *new elements*, while a list is closed: if imported in a package, it cannot be modified in the context of the importer package. Thus, a set of reusable labels (especially in combination with corresponding views) can be reused in several packages, as a description schema (but with a more flexible structure).

Generally, a package can associated any tag (own or imported) to any element (own or imported). A package *p1* could even import from the *p2* package a *t* tag and a *e* element, and associate them, even it *t* and *e* are not associated in their origin package *p2*.

It is thus crucial to understand that the tag relationship is in fact a **ternary** relation between a tag, the associated element *and* the package defining the association. Associations defined by a package are automatiquement imported with the package (as any package element). Thus, any package "inherits" the associations defined by the package it imports.

A tag can implement the Group interface *only* in the context of a specific package (for instance the currently edited package).

4.2.12 Pre-defined groups

A package always possesses two pre-defined groups named *own* and *all*. The *own* group contains, by definition, all elements defined within the package. The *all* group contains, by definition, all element defined and imported (directly or not) by the package. It is consequently the union of the *own* group with all dynamically imported elements.

These groups cannot be removed from the package.

CHAPTER

FIVE

CINELAB APPLICATION MODEL - CAM

In this document, QNames are used to abbreviate URLs (by concatenating the namespace with the suffix). We use the following namespaces:

- @prefix dc: http://purl.org/dc/elements/1.1/
- @prefix cam: http://advene.org/ns/cinelab/
- @prefix pm: http://advene.org/ns/parser-meta/

5.1 Generalities

The Cinelab application model defines some higher-level concepts such as annotation types, relation types and schemas, relying on the data model defined in the hypervideo model. The following figure gives an overview of these new element types. It also specifies more precisely some metadata associated to elements of the model.

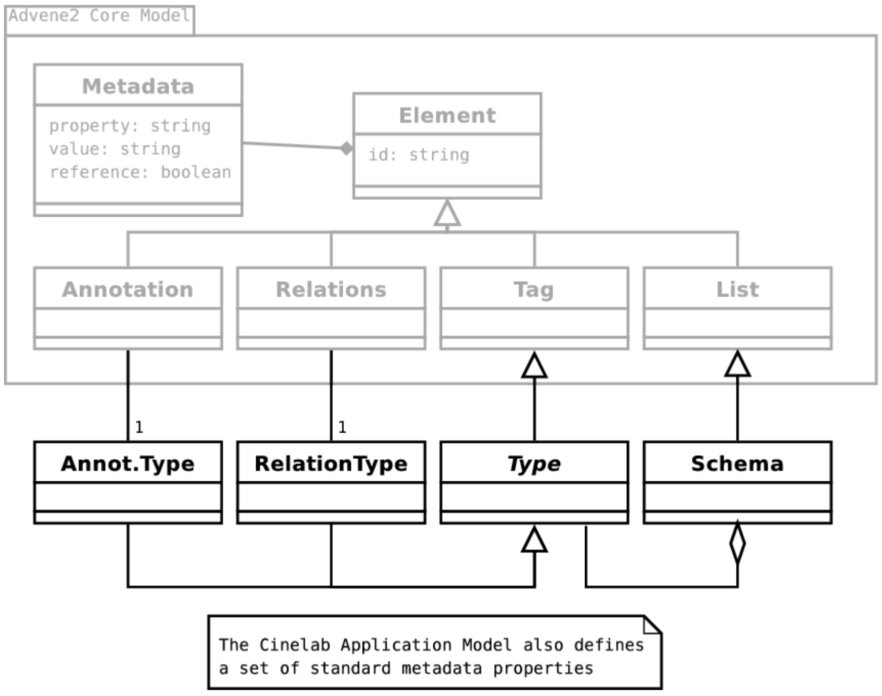

Figure 5.1: Cinelab specific elements

5.1.1 Package

In order to ease elements addressing, package elements SHOULD be addressable through the TALES syntax as in the current Advene version. This syntax offers a uniform access to the data model, that can be stored in various ways (XML file, database, etc). Moreover, it allows to use TALES expressions as URL parts, easing the development of REST-based architectures.

Any TALES expression must be evaluated in a given context. This context MUST reference a specific package, named *reference package*. It MUST be accessible in TALES through the named `refpackage`. When the TALES expression is used inside of an element content, the reference package is the package that owns the element.

To make them easier to identify, and ease interoperability with other applications, we define a *TALES string* as a character string that can contain TALES expressions identified with the notation `${...}`. This makes it possible to gracefully degrade TALES expression support: applications without TALES support will display the expression, clearly identified. TALES-supporting systems only have to systematically preprend the `string:` prefix to the character string before sending it to the TALES evaluator.

Metadata

Meaning	Element	API property
prefix/namespace association	pm:namespaces	namespaces

The `pm:namespaces` metadata is used to simplify URI use, especially for metadata. Its structure is a set of couples (prefix, namespace URI). Each couple is encoded by concatenating the prefix and its URI, separated with a space. Couples are separated by a newline. Using this information, any time a URI is used, the user interface can replace it with a RDF-like QName.

NB: this metadata belongs to a specific namespace, because it has to be specifically processed when seralising/analysing a Cinelab XML package. Instead of being stored as other element metadata, it is encoded/fetched from the XML namespace declarations (*xmlns* attributes).

5.1.2 Elements

5.1.3 Common metadata

Metadata keys MUST be URIs.

The package and any package element MUST possess the following Dublin Core metadata:

Meaning	Element	API property
Creator	dc:creator	creator
Last contributor	dc:contributor	contributor
Creation date	dc:created	created
Last modification date	dc:modified	modified

Dates MUST be encoding according the ISO 8601 norm, with restrictions specified in http://www.w3.org/TR/NOTE-datetime .

Optionaly, the following metadata MAY be defined:

Meaning	Element	API property
Title	dc:title	title
Description	dc:description	description

TALES Addressing

Any object with metadata MUST expose a TALES attribute `meta` that offers an access to its metadata through a TALES expression like `obj/meta/dc/creator`

where `dc` (in the example) is a prefix declared in the `pm:namespaces` metadata. More precisely, the `pm:namespaces` of the *reference package* for the TALES expression will be used (and not the package owning the element).

Metadata defined by the applicative model can specify a synonym property in the API column, for instance `creator` for `dc:creator`. In this case, an equivalent TALES expression can be `obj/creator`.

5.1.4 Content

A content specifies its MIME type.

Some generic MIME types like `application/xml` or `application/json` may specify additional contraints for some content sub-types (as XML schemas or JSON-schemas for instances). A content MAY specify a *model* attribut that will contain the identifier of a resource whose content will be the appropriate schema.

The following schemas MAY be implemented:

Schema	Content MIME type	Schema MIME type	Schema content
Perl Compatible Regular Expression (PCRE)	text/*	application/x-advene-schema-pcre	A regular expression
RelaxNG	application/xml	application/relax-ng-compact-syntax	A RelaxNG schema in compact notation

5.2 Elements

5.2.1 Media

Attributes

Meaning	API property
URL	url
unit (ms, frame...)	unit
origin (in specified unit)d	origin

The only temporal reference that an implementation MUST support uses the “ms” unit (milliseconds), with a 0 origin.

A URL is used to specify the designated media. In the specific case of DVDs, there is no standard to define generic URLs. The model specifies the following format: `dvd://title/chapter` where `title` and `chapter` are optional numbers (defaulting to 1 if not specified). Application should convert this URL to the appropriate format for the used multimedia player.

Metadata

The following metadata MAY be defined:

Meaning	Element	API property
Resource duration	cam:duration	duration
URI	cam:uri	uri

If the URL is not accessible, the URI is used as a key to identify another URL for the same media.

5.2.2 Test

A test is a view that returns a value interpretable as a boolean value. It is notably used to express constraints on elements.

The following MIME type MAY be implemented:

`application/x-advene-builtin-test`	Test défini en interne par Advene

The *builtin* test has for content a number of attributes to check. For instance, `content-type=text/html`. Such a format has the interesting property of being easily introspected, in order to allow automatic GUI generation.

5.2.3 Annotation

Metadata

The following metadata MUST be specified.

Meaning	Element	API property
Annotation type id	cam:type	type

The following metadata MAY be defined:

Meaning	Element	API property
Annotation color	cam:color	TALES string that can be evaluated in the annotation context to get its color

5.2.4 Relation

Metadata

The following metadata MUST be specified.

Meaning	Element	API property
Relation type id	cam:type	type

The following metadata MAY be defined:

Meaning	Element	API property
Relation color	cam:color	TALES string that can be evaluated in the relation context to get its color

5.2.5 Annotation type and relation type

An annotation type (resp. a relation type) is a tag, that has the `cam:system-type` metadata, with value "annotation-type" (resp. "relation-type"). An annotation (resp. relation) MUST be associated with exactly one annotation type (resp. relation type), and this association MUST be consistent with the annotation (resp. relation) metadata `cam:type`.

Metadata

The following metadata MAY be defined:

Meaning	Name	Content
Element representation	cam:representation	TALES string that should be evaluated on elements of the type, to get a compact textual representation
Type color	ca:color	TALES string that should be evaluated on the type to get its color.
Element color	cam:element-color	TALES string that should be evaluated on elements of the type, to get their color
Generic constraint	cam:element-constraint	Test view that elements of this type should match
Content type constraint [1]	cam:content-mimetype	String specifying the MIME type for elements of the type
Content model constraint [1]	cam:content-model of the type	Resource specifying the content model for elements of the type

5.2.6 Schema

A schema is a list whose metadata `cam:system-type` value is "schema". A schema represents a description schema, grouping a consistent set of annotation types, relation types and possibly other elements related to the description choices (resources containing content models, tests, associated views...).

5.2.7 Tag

Metadata

The following metadata MAY be defined:

Meaning	Element	API property
Tag color	cam:color	TALES string that can be evaluated in the tag context to get its color

5.2.8 Resources

Core model resources. Some may be hierarchised as in a filesystem. Their identifier will then be prefixed with `:userfile:` and will encode their path in the identifier by separating directory names with `:`.

5.2.9 Implementation remarks

Any python object that will have to be handled in the Advene model has to be wrapped in a (temporary) Resource object, so that model consistency is preserved. For efficiency reasons, effective wrapping is carried out lazily.

For instance, a view produces a resource. But if is is included in another view (through TAL for instance), there is no need to transform it into a resource, since it will be consumed (by the including view) before reaching the Advene model level. The including view has to produce a resource, which will be returned to the application.

[1] both metadata (Content type constraint and content model constraint) bring no more expressivity than Generic constraint (which is more generic), but they allow to remove one indirection level (the Test view) for most frequently used constraints.

5.2.10 Dynamic imports

Attributes

Meaning	API property
URL	url
URI	uri

If an URL is not accessible, the URI value can be used by the application to identify another URL for the same package.

If the package loaded at a given URL declares a URI other than specified by the dynamic import, the application SHOULD notify the user to determine the appropriate way to handle this inconsistency (update the dynamic import, or look for the given URI at another URL).

CHAPTER

SIX

CINELAB FILE FORMAT

6.1 Introduction

The data model defined in the previous sections must be serialized to be exchangeable. To this effect, 2 serialization formats have been specified, in order to improve interoperability between applications that use the Cinelab data model. The first format is based on XML, while the second one uses the JSON syntax. In addition, a Zip-based serialization has been specified in order to make it more convenient to store huge, non-text resources.

6.2 File extensions

To ease identification of Cinelab package, applications SHOULD use the following file extensions: `.cxp` for plain XML files (Cinelab XML Package), `.czp` for compressed files (Cinelab Zip Package), `.cjp` for JSON files (Cinelab JSON Package).

6.3 Cinelab Zip-Packages

XML does not offer a standard way to correctly handle large binary objects like images, application files, etc. Moreover, plain XML files can reach huge sizes. The same arguments apply to JSON syntax. We thus use a OpenDocument-like format to store the XML representation of a package with its associated binary files, and to compress this content. The file is a standard Zip file, whose structure is described below.

Information about the files present in the package is stored in a XML *manifest* file. It is always stored as `META-INF/manifest.xml`. Its main data is

- a list of all package files
- the MIME type for each file
- if one of the files is encrypted, the necessary information to allow its decryption.

6.3.1 General layout

The package is serialized as a .zip file, using the same layout and principles as the OpenDocument format (see pp. 684-692 of http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf).

General layout:

- `mimetype` : the MIME type (`application/x-advene-zip-package`)

- `content.xml` : le plain XML package
- `userfiles/` : a file hierarchy (accessible through the `package/resources` TALES path), containing any file (CSS, icons...) necessary to build views from the package. From the core model point of view, each file is a resource, whose id has the `:user_file:` prefix, and encodes the path by separating directory names with `:`. NB: directories will also appear as resources, of a specific type `inode/directory`.
- `data/` : internal data associated to the package (rich/externalized annotation content)
- `preview.xml` : aggregated statistical data, to ease previews/searches

Contents (for annotations, relations, views, etc) can either be stored directly in the XML file, or externalized in the `data/` directory. (cf OpenDocument p. 686)

In a given file contained in a package, relative URIs are used to reference other files of the same package, but also to reference other files of the filesystem. The following restrictions are imposed for internal references: * only files of the same package can be referenced internally * URIs referencing another file of the same package MUST be relative and MUST NOT contain paths that are not part of the package. This notably means that files in a package MUST NOT be referenced through an absolute URI. * a file in a package cannot be referenced from the outside of the package (either from the filesystem or another package)

A relative path present in a file contained in a package must be parsed exactly like it would if the package is uncompressed in a directory with the same basename as the package. The base URI of relative path is the URI of the directory containing the file containing the relative path.

For instance, the `userfiles/foo.txt` references a user file (package resource). `../file.txt` allows to access a file in the same directory as the package.

Any other URI reference, specifically those that specify a protocol (http:), an authority (i.e. //) or an absolute path (i.e. /) do not need any specific processing. This means that absolute paths do not reference files inside of the package, but inside of the hierarchy (filesystem most of the time) containing the package.

6.3.2 Thumbnails

A graphical, iconic representation of the document MAY be generated when the file is saved. It should be a representation of the default view for the packagem, and should be generated without effect, frame or borders.

The icon is saved as `Thumbnails/thumbnail.png`. The file and containing directory are not mentioned in the `manifest.xml` file, since they are not really part of the document.

In accordance with the *Thumbnail Managing Standard* (TMS) (cf www.freedesktop.org), icons MUST be saved as 24-bit PNG files, non-interlaces, with complete alpha transparency. The required size is 128x128 pixels.

6.3.3 Manifest file

Cf OpenDocument spec, p. 687

6.4 XML serialization

6.4.1 Encoding

The encoding of XML serialisation MUST be UTF-8.

6.4.2 Metadata

In accordance with the model, package metadata MUST contain the following keys: `dc:creator`, `dc:created`, `dc:contributor`, `dc:contributed`. In package elements, these metadata may be omitted from the serialisation, and are then inherited (since they must be available in the model) using the following rules:

- `dc:creator`, `dc:created`: the element inherits the value from its package
- `dc:contributor`: if the `dc:creator` is explicitly specified for the element, its value is used; else, the `dc:contributor` package value is used.
- `dc:modified`: if the `dc:created` is explicitly specified for the element, its value is used; else, the `dc:modified` package value is used.

In the example XML file, multiple commented cases are proposed.

6.4.3 Namespaces

The package `pm:namespaces` metadata is specifically processed: it is encoded in the XML root element as `xmlns` attributes.

6.4.4 Type declaration

To make the generated XML easier to read, some metadata specified in the applicative model are encoded as attributes instead of plain metadata (`type` for annotations and relations), or as elements (*annotation-type*, *relation-type*, *schema*). See the RelaxNG below for more information.

6.4.5 RelaxNG schema

The compact RelaxNG notation is used to specify the proposed format: cinelab.rnc

```
# Advene XML format description
# RNC Tutorial: http://relaxng.org/compact-tutorial-20030326.html

# Cinelab Application Model:
default namespace = "http://advene.org/ns/cinelab/"
namespace cam = "http://advene.org/ns/cinelab/"

# Dublin Core model
namespace dc = "http://purl.org/dc/elements/1.1/"

# XML Schema datatypes
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"

grammar {
   start = element package {
      ## id_attribute &
      ##   Not really an ID, but something is needed to *suggest* ids for
      ##   dynamic imports or application identifier (for the web server).
      ##   Should probably be dynamically extracted from the URI instead.
      uri_attribute?
    &
      element medias {
         element media {
            id_attribute &
            url_attribute &
            attribute unit { "ms" | "frame" }? & # defaults to ms
            attribute origin { xsd:long }?  &    # defaults to 0
            tags_element? &
            element meta {
               common_meta_elements &
               element duration { xsd:long }? &
               uri_meta_element?
            }?
         }*
      }?
    &
      element imports {
         element import {
            attribute id { import-identifier } &
            url_attribute &
            uri_attribute? &
            tags_element? &
            element meta {
               common_meta_elements
            }?
         }*
      }?
    &
      element annotations {
         element annotation {
            id_attribute &
            attribute media { identifier-ref } &
            attribute begin { xsd:long } &
            attribute end { xsd:long } &
            content_element &
            tags_element? &
            element meta {
               common_meta_elements &
               element type { id-ref_attribute }
            }?
         }*
      }?
    &
      element relations {
         element relation {
            id_attribute &
            content_element? &
            element members {
               element member { id-ref_attribute }*
            } &
            tags_element? &
            element meta {
               common_meta_elements &
               element type { id-ref_attribute }
            }?
         }*
      }?
    &
      element tags {
         element tag {
            id_attribute &
            tags_element? &
            element imported-elements {
               element \element { id-ref_attribute }*
            }? &
            element meta {
               common_meta_elements &
               constraint_meta_element?
            }?
         }*
      }?
    &
      element annotation-types {
         element annotation-type { type_structure }*
      }?
    &
      element relation-types {
         element relation-type { type_structure }*
      }?
    &
      element lists {
         element \list { list_structure }*
      }?
    &
      element schemas {
         element schema { list_structure }*
      }?
    &
      element queries {
         element query {
            id_attribute &
            content_element &
            tags_element? &
            element meta {
               common_meta_elements &
               constraint_meta_element?
            }?
         }*
      }?
    &
      element views {
         element view {
            id_attribute &
            content_element &
            tags_element? &
            element meta {
               common_meta_elements &
               constraint_meta_element?
            }?
         }*
      }?
    &
      element resources {
         element resource {
            id_attribute &
            content_element &
            tags_element? &
            element meta { common_meta_elements }?
         }*
      }?
```

```
 &
   element external-tag-associations {
      element association {
         attribute \element { identifier-ref } &
         attribute tag { identifier-ref }
      }*
   }?
 &
   element meta {
      element dc:creator { text } &
      element dc:contributor { text } &
      element dc:created { xsd:dateTime } &
      element dc:modified { xsd:dateTime } &
      element * -
      (dc:creator | dc:contributor | dc:created | dc:modified | cam:*) {
         id-ref_attribute | text
      }*
   }
}



##
## Reusable elements and structures
##

## tags_element is used in all elements.
tags_element = element tags {
   element tag { id-ref_attribute }*
}

## content_element is used in annotation, relation, queries, views &
## resources.
## A content has a mimetype, and defines its data either through a
## reference to an external resource, or through its #DATA section.
content_element = element content {
   attribute mimetype { text }?, # defaults to text/plain
   attribute encoding { "base64" }?, # only encoding supported
   (url_attribute | text)
}

## type_structure defines the common structure of the following elements:
## annotation-type, relation-type
type_structure =
   id_attribute &
   tags_element? &
   element meta {
      common_meta_elements &
      constraint_meta_element? &
      element representation { TALESstring }? &
      element element-color { TALESstring }? &
      element content-mimetype { text }? &
      element content-model { id-ref_attribute }?
   }

## list_structure defines the common structure of the following elements:
## list, schema
list_structure =
   id_attribute &
   element items {
      element item { id-ref_attribute }*
   }? &
   tags_element? &
   element meta {
      common_meta_elements
    &
      constraint_meta_element?
   }



##
## meta-data related elements and structures
##

common_meta_elements =
   element dc:creator { text }? &
   element dc:contributor { text }? &
   element dc:created { xsd:dateTime }? &
   element dc:modified { xsd:dateTime }? &
   # Almost any element can define a color
   element color { TALESstring }? &
   # Allow to have user-defined metadata items
   element * - (dc:creator | dc:contributor | dc:created | dc:modified |
                cam:*) {
      id-ref_attribute | text
   }*

constraint_meta_element = element element-constraint {
   # Reference an existing view/test
   id-ref_attribute
}

uri_meta_element = element uri { xsd:anyURI }



##
## reusable attributes
##

## NB: it would have been sensible to use 'href' instead of 'url', which is
## common practice to hold a link (HTML, XLink), but since the API uses
## 'url', it seemed a better idea to keep the XML format consistent with the
## URI
url_attribute =  attribute url { xsd:anyURI }
uri_attribute =  attribute uri { xsd:anyURI }

id_attribute = attribute id { identifier }
id-ref_attribute = attribute id-ref { identifier-ref }



##
## attribute special datatypes
##
```

```
   ## Naming identifiers
   identifier = xsd:string {
      pattern = "([a-zA-Z_][a-zA-Z0-9_\-]*|:[a-zA-Z0-9_:\-]*)"
   }

   ## import identifiers have additional restrictions restrictions
   import-identifier = xsd:ID { pattern = "[a-zA-Z_][a-zA-Z0-9_\-]*" }

   ## Identifier references allow linked elements (beginning with : )
   identifier-ref = xsd:string {
      pattern = "([a-zA-Z_:][a-zA-Z0-9_:\-]*)"
   }

   ## a regexp for TALESstrings would be overly complex, so: A
   ## TALES-string is a character string which can contain TALES
   ## expressions embedded with the ${...} notation. For example, the
   ## TALES expression foo/bar alone is represented by the TALES
   ## string ${foo/bar}
   TALESstring = text
}
```

6.4.6 Example XML file

An example of conforming XML is given below, and can be downloaded here.

```
<package xmlns="http://advene.org/ns/cinelab/" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <meta>
    <dc:creator>pchampin</dc:creator>
    <dc:created>2010-09-01T12:33:53.403508</dc:created>
    <dc:contributor>oaubert</dc:contributor>
    <dc:modified>2010-09-06T12:33:53.420459</dc:modified>
    <dc:description>Example Cinelab package</dc:description>
    <dc:title>Nosferatu analysis</dc:title>
    <default_utbv xmlns="http://www.advene.org/ns/advene/">start_view</default_utbv>
  </meta>
  <imports>
    <import id="cam" url="http://liris.cnrs.fr/advene/cam/bootstrap" />
  </imports>
  <annotation-types>
    <annotation-type id="free-text-annotation">
      <tags>
        <tag id-ref="important" />
        <tag id-ref="todo" />
      </tags>
      <meta>
        <dc:modified>2010-09-02T12:33:53.416368</dc:modified>
        <!-- dc:creator, dc:created are inherited from the package -->
        <!-- dc:contributor is inherited *from the package*, as no dc:creator
             is explicitly specified here -->
        <color>#55ff55</color>
        <element-color>${here/tag_color}</element-color>
        <element-constraint id-ref=":constraint:free-text-annotation" />
        <dc:description>Shot layout of the movie</dc:description>
        <dc:title>Shots</dc:title>
      </meta>
    </annotation-type>
```

```xml
    <annotation-type id="shots">
      <meta>
        <dc:created>2010-09-02T12:33:53.414772</dc:created>
        <dc:creator>oaubert</dc:creator>
        <!-- dc:contributor, dc:modified are inherited from dc:creator and
             dc:created, respectively -->
        <element-constraint id-ref=":constraint:shots" />
      </meta>
    </annotation-type>
  </annotation-types>
  <tags>
    <tag id="important">
      <meta>
        <color>#00ff00</color>
        <dc:created>2010-09-02T12:33:53.407836</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-03T12:33:53.409026</dc:modified>
        <!-- dc:contributor is inherited from dc:creator -->
        <dc:description>Important things to note</dc:description>
        <dc:title>Important</dc:title>
      </meta>
    </tag>
    <tag id="todo">
      <meta>
        <dc:contributor>pchampin</dc:contributor>
        <dc:modified>2010-09-03T12:33:53.406964</dc:modified>
        <!-- dc:creator and dc:created are inherited from the package -->
        <color>#ff4444</color>
        <dc:description>Things to work on</dc:description>
        <dc:title>TODO</dc:title>
      </meta>
    </tag>
  </tags>
  <medias>
    <media id="m1" url="/data/video/Nosferatu.avi" origin="0" unit="ms">
      <meta>
        <uri>http://liris.cnrs.fr/advene/videos/baz.avi</uri>
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.404347</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.404904</dc:modified>
      </meta>
    </media>
  </medias>
  <annotations>
    <annotation begin="1230" end="4560" id="a1" media="m1">
      <content mimetype="text/plain">{ ’num’ : 1, ’title’: ’Introduction’, ’characters’: [ ’john doe’
      <meta>
        <type id-ref="free-text-annotation" />
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.417550</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.420459</dc:modified>
      </meta>
    </annotation>
    <annotation begin="1230" end="4560" id="a3" media="m1">
      <content encoding="base64" mimetype="application/json" />
      <meta>
```

```
        <type id-ref="shots" />
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.419975</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.419975</dc:modified>
      </meta>
    </annotation>
    <annotation begin="4560" end="7890" id="a2" media="m1">
      <content encoding="base64" mimetype="image/png" />
      <meta>
        <type id-ref="free-text-annotation" />
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.418975</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.418975</dc:modified>
      </meta>
    </annotation>
  </annotations>
  <views>
    <view id=":constraint:free-text-annotation">
      <content mimetype="application/x-advene-type-constraint">mimetype=application/json</content>
      <meta>
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.410127</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.416718</dc:modified>
      </meta>
    </view>
    <view id=":constraint:shots">
      <content mimetype="application/x-advene-type-constraint" />
      <meta>
        <dc:contributor>oaubert</dc:contributor>
        <dc:created>2010-09-06T12:33:53.414208</dc:created>
        <dc:creator>oaubert</dc:creator>
        <dc:modified>2010-09-06T12:33:53.414208</dc:modified>
      </meta>
    </view>
  </views>
</package>
```

6.5 JSON serialization

The JSON serialization has been defined to facilitate the exchange of package information in web-based contexts.

6.5.1 Encoding

The encoding of JSON serialisation MUST be UTF-8.

6.5.2 Type declaration

To make the generated JSON easier to read, some metadata specified in the applicative model are encoded as attributes instead of plain metadata (`type` for annotations and relations), or as elements (*annotation-type*, *relation-type*, *schema*).

6.5.3 General layout

The package is represented by a JSON object with the following properties:

- `format`: always `"http://advene.org/ns/cinelab/"`
- each of the following property will reference an array of JSON objects: `imports`, `medias`, `annotations`, `relations`, `tags`, `annotation_types`, `relation_types`, `lists`, `schemas`, `queries`, `views`, `resources`
- for every element (the top-level package, and all defined model elements), an associated `meta` object holds its metadata, model-defined and user-defined.

6.5.4 Metadata

In accordance with the model, package metadata MUST contain the following keys: `creator`, `created`, `contributor`, `contributed`. In package elements, these metadata may be omitted from the serialisation, and are then inherited (since they must be available in the model) using the following rules:

- `creator`, `created`: the element inherits the value from its package
- `contributor`: if the `creator` is explicitly specified for the element, its value is used; else, the `contributor` package value is used.
- `modified`: if the `created` is explicitly specified for the element, its value is used; else, the `modified` package value is used.

The following example JSON file provides an example package.

```
{
    "format": "http://advene.org/ns/cinelab/",
    "imports": [{
        "id": "acav",
        "url": "http://acav.dailymotion.com/std-schemas-v1.cjp"
    }],
    "medias": [{
        "id": "video",
        "url": "http://www.dailymotion.com/video/xdg0h0",
        "meta": {
            "title": "Ben se fait des films"
        }
    }],

    "annotation_types": [
        {
            "id": "Character",
            "meta": {
                "description": "Appearance of the main characters.",
                "content-mimetype": "application/json",
                "content-model": { "id_ref": "Characters_model" }
            }
        },
        {
            "id": "Supernatural",
            "meta": {
                "description": "An appearance of something supernatural."
            }
        }
    ],
    "resources": [
        {
            "id": "Character_model",
            "content": {
                "data": {
                    "enum": ["Dracula", "Jonathan", "Nina", "Reinfield"]
                }
            }
        },
    ],

    "tags": [
        {
            "id": "funny"
        },
        {
            "id": "scary"
        }
    ],

    "annotations": [
        {
            "id": "a1",
            "type": "Supernatural",
            "media": "video",
            "begin": 1234,
            "end": 5678,
            "content": {
                "data": "a flying toaster"
            },
            "tags": [ "funny" ]
        },
        {
            "id": "a2",
            "type": "acav:TimedText",
            "media": "video",
            "begin": 1234,
            "end": 5678,
            "content": {
                "mimetype": "application/json",
                "data": {
                    "text": "ceci est un sous-titre",
                    "style": "font-size: 120%"
                },
                "model": "acav:TimedText_model"
            },
            "tags": [ "funny", "scary" ]
        },
        {
            "id": "a3",
            "type": "Character",
            "media": "video",
            "begin": 234,
            "end": 567,
            "content": {
                "data": '"Nina"'
            },
            "tags": [ "funny" ]
        },
    ],

    "meta": {
        "creator": "Pierre-Antoine Champin",
        "created": "2011-06-09T07:25:43",
        "contributor": "Pierre-Antoine Champin",
        "modified": "2011-06-09T07:25:43"
    }
}
```

6.5.5 JSON-Schema

Two JSON-Schema schemas are proposed: a general schema and a more strict schema that does not allow additional undefined properties to be added to elements.

We include below the more permissive schema:

```
{
    "description":"Cinelab JSON Package (CJP)",
    "version": "1.0",
    "$schema" : "http://json-schema.org/draft-03/schema#",
    "id" : "http://advene.org/ns/cinelab/cjp#",
    "type":"object",

    "properties":{

        "__definitions": {
            "description": "A placeholder for reusable subschemas",
            "type": [
                {
                    "id": "#id_ref",
                    "type": "string",
                    "pattern": "^([a-zA-Z_][a-zA-Z0-9_\\-]*:)?([a-zA-Z_][a-zA-Z0-9_\\-]*|:[a-zA-Z0-9_
                },
                {
                    "id": "#strict-id_ref",
                    "type": "string",
                    "pattern": "^[a-zA-Z_][a-zA-Z0-9_\\-]*:([a-zA-Z_][a-zA-Z0-9_\\-]*|:[a-zA-Z0-9_:\
                },
                {
                    "id": "#TALESstring",
                    "type": "string",
                    "description": "a regexp for TALESstrings would be overly complex, so: A TALES-st
                },
                {
                    "id": "#proto-content",
                    "type": "object",
                    "properties": {
                        "mimetype": {
                            "description": "TODO: better regex for mimetypes?",
                            "type": "string",
                            "pattern": "^[-+a-z0-9]+/[-+a-z0-9]+$",
                            "default": "text/plain"
                        },
                        "model": { "$ref": "#id_ref" }
                    }
                },
                {
                    "id": "#content-with-data",
                    "extends": [{ "$ref": "#proto-content" }],
                    "properties": {
                        "url": {
                            "type": "string",
                            "format": "uri",
                            "required": true
                        },
                        "data": {
                            "disallow": "any"
                        },
                        "encoding": {
                            "disallow": "any"
                        }
                    }
                },
                {
                    "id": "#content-with-url",
                    "extends": [{ "$ref": "#proto-content" }],
                    "properties": {
                        "data": {
                            "type": ["string", "object"],
                            "required": true
                        },
                        "encoding": {
                            "enum": ["base64"]
                        },
                        "url": {
                            "disallow": "any"
                        }
                    }
                },
                {
                    "id": "#content",
                    "type": [
                        { "$ref": "#content-with-url" },
                        { "$ref": "#content-with-data" }
                    ]
                },
                {
                    "id": "#meta",
                    "type": "object",
                    "properties": {
                        "creator": {
                            "type": "string"
                        },
                        "created": {
                            "type": "string",
                            "format": "date-time"
                        },
                        "contributor": {
                            "type": "string"
                        },
                        "modified": {
                            "type": "string",
                            "format": "date-time"
                        }
                    },
                    "additionalProperties": {
                        "type": ["object", "string", "number", "boolean"],
                        "properties": {
                            "id_ref": { "$ref": "#id_ref" }
                        },
                    }
                },
                {
                    "id": "#element",
                    "type": "object",
                    "properties": {
                        "id": {
                            "type": "string",
                            "pattern": "^([a-zA-Z_][a-zA-Z0-9_\\-]*|:[a-zA-Z0-9_:\\-]*)$",
                            "required": true
                        },
                        "tags": {
                            "type": "array",
                            "items": { "$ref": "#id_ref" }
                        },
                        "meta": {
                            "extends": [{ "$ref": "#meta" }],
                            "properties": {
                                "color": { "$ref": "#TALESstring" }
                            }
                        }
                    }
                },
                {
                    "id": "#element-with-content",
                    "extends": [{ "$ref": "#element" }],
                    "properties": {
                        "content": {
                            "extends": [{ "$ref": "#content" }],
                            "required": true
                        }
                    }
                },
                {
                    "id": "#type",
                    "extends": [{ "$ref": "#element" }],
                    "properties": {
                        "meta": {
                            "properties": {
                                "content_mimetype": { "type": "string" },
                                "content_model": { "type": "object" },
                                "element_constraint": { "type": "object" },
                                "representation": { "$ref": "#TALESstring" },
                                "elementColor": { "$ref": "#TALESstring" }
                            }
                        }
                    }
                },
                {
                    "id": "#list",
                    "extends": [{ "$ref": "#element" }],
                    "properties": {
                        "items": {
                            "type": "array",
                            "items": { "$ref": "#id_ref" }
                        },
                        "meta": {
                            "properties": {
                                "element_constraint": { "type": "object" }
                            }
                        }
                    }
                }
            ]
        },


        "format": {
            "enum": [ "http://advene.org/ns/cinelab/" ],
            "required": true
        },

        "@context": {
            "type": "object",
            "patternProperties": {
                "[a-zA-Z_][a-zA-Z0-9-_.]*": {
                    "type": "string",
                    "format": "uri",
                }
            }
        },

        "@": {
            "type": "string",
            "format": "uri",
        },

        "imports": {
            "type": "array",
            "items": {
                "extends": [{ "$ref": "#element" }],
                "properties": {
                    "id": {
                        "type": "string",
                        "pattern": "^[a-zA-Z_][a-zA-Z0-9_\\-]*$",
                    },
                    "url": {
                        "type": "string",
                        "format": "uri",
                        "required": true
                    },
                    "uri": {
                        "type": "string",
                        "format": "uri"
                    }
                }
            }
        },
        "medias": {
            "type": "array",
            "items": {
                "extends": [{"$ref": "#element"}],
                "properties": {
                    "url": {
                        "type": "string",
                        "format": "uri",
                        "required": true
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["ms", "frame"],
                        "default": "ms"
                    },
                    "origin": {
                        "type": "integer",
                        "default": 0
                    },
                    "meta": {
                        "properties": {
                            "duration": {
                                "type": "integer"
                            },
                            "uri": {
                                "type": "string",
                                "format": "uri"
                            }
                        }
                    },
                    "frame_of_reference": {
                        "type": "string",
                        "format": "uri"
                    }
                }
            }
        },

        "annotations": {
            "type": "array",
            "items": {
                "extends":  [{ "$ref": "#element-with-content" }],
                "properties": {
                    "type": {
                        "extends": [{ "$ref": "#id_ref" }],
                        "required": true
                    },
                    "media": {
                        "extends": [{ "$ref": "#id_ref" }],
                        "required": true
                    },
                    "begin": {
                        "type": "integer",
                        "required": true
                    },
                    "end": {
                        "type": "integer",
                        "required": true
                    }
                }
            }
        },

        "relations": {
            "type": "array",
            "items": {
                "extends":  [{ "$ref": "#element" }],
                "properties": {
                    "type": {
                        "extends": [{ "$ref": "#id_ref" }],
                        "required": true
                    },
                    "members": {
                        "type": "array",
                        "items": { "$ref": "#id_ref" }
                    },
                    "content": { "$ref": "#content" }
                }
            }
        },

        "tags": {
            "type": "array",
            "items": {
                "extends":  [{ "$ref": "#element" }],
                "properties": {
                    "imported_elements": {
                        "type": "array",
                        "items": { "$ref": "#strict-id_ref" }
                    },
                    "meta": {
                        "properties": {
                            "element_constraint": { "type": "object" }
                        }
                    }
                }
            }
        },

        "annotation_types": {
            "type": "array",
            "items": { "$ref": "#type" }
        },

        "relation_types": {
            "type": "array",
            "items": { "$ref": "#type" }
        },

        "lists": {
            "type": "array",
            "items": { "$ref": "#list" }
        },

        "schemas": {
            "type": "array",
            "items": { "$ref": "#list" }
        },

        "queries": {
            "type": "array",
            "items": {
                "extends": [{"$ref": "#element-with-content"}],
                "properties": {
                    "meta": {
                        "properties": {
                            "element_constraint": { "type": "object" }
                        }
                    }
                }
            }
        },

        "views": {
            "type": "array",
            "items": {
                "extends": [{ "$ref": "#element-with-content" }],
                "properties": {
                    "meta": {
                        "properties": {
                            "element_constraint": { "type": "object" }
                        }
                    }
                }
            }
        },

        "resources": {
            "type": "array",
            "items": { "$ref": "#element-with-content" }
        },

        "tagging": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "element": {
                        "extends": [{ "$ref": "#strict-id_ref" }],
                        "required": true
                    },
                    "tag": {
                        "extends": [{ "$ref": "#strict-id_ref" }],
                        "required": true
                    }
                }
            }
        },

        "meta": {
            "extends": [{ "$ref": "#meta" }],
            "creator": { "required": true },
            "created": { "required": true },
            "contributor": { "required": true },
            "modified": { "required": true }
        }
    }
}
```