ISO/IEC JTC 1/SC 29/WG 03 N1348

**ISO/IEC JTC 1/SC 29/WG 03  
MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title:** Technology under Consideration on ISO/IEC 14496-34 Syntactic Description Language

**Status:** Approved

**Date of document:** 2024-11-28

**Source:** ISO/IEC JTC 1/SC 29/WG 03

**No. of pages:** 12 (with cover page)

**Email of Convenor:** young.L @ samsung . com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N1348**

**November 2024, Kemer, TR**

|  |  |
| --- | --- |
| **Title** | **Technology under Consideration on ISO/IEC 14496-34 Syntactic Description Language** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **24411** |

# Introduction

This document collects technologies being under study for consideration in the development of the standard ISO/IEC 14496-34 Syntactic Description Language

# Table of content

[1. Introduction 1](#_Toc183699389)

[2. Table of content 1](#_Toc183699390)

[3. Grammar file for SDL syntax 1](#_Toc183699391)

[4. On the keyword template 4](#_Toc183699392)

[5. On optional class member 4](#_Toc183699393)

# Grammar file for SDL syntax

## General

This clause provides the SDL grammar implementing the rules of the present specification. The grammar is based on the parsing expression grammar (PEG) concept and following the syntax and set of rules defined by the Pegen software project [1].

Currently, the grammar file is hosted an developed at <http://mpegx.int-evry.fr/software/MPEG/Systems/sdl/sdl-grammar>.

The usage of PEG seems adequate for the validation of the SDL syntax, however there are still some questions whether the PEG grammar defined by the Pegen software project is appropriate for the developing the conformance of the SDL specification. Further study is encourage on other possible alternative for such grammar file.

## Grammar file

[Editor’s note] The following is in work-in-progress and may not reflect all the rules described in the latest SDL specification text.

1. SDL grammar file

|  |
| --- |
| start: file\_input  file\_input: (NEWLINE+ | line)\* ENDMARKER  line: class\_def | comment\_cpp  #NOTE: 14496-1 forbids going back to a new line berfore {  # Rule C.1 and C.2  class\_def: aligned? abstract? 'class' NAME+ parameter\_list? parent\_class? NEWLINE? '{' body? '}'  aligned: 'aligned' '(' NUMBER+ ')'  abstract: 'abstract'  parameter\_list: '(' ','.parameter+ ')'  # NOTE: we allow arrays to be passed as parameter class, however 14496-1 is not clear on this  parameter: optional? type NAME array\_length?  # NOTE: Not in 14496-1, not sure where this come from  optional: 'optional'  # NOTE: would allow "unsigned bit", to be improved  type: signed? data\_type  signed: 'unsigned'  data\_type:  | 'bit'  | 'int'  | 'double'  array\_length: '[' NUMBER\* ']'  #NOTE: 14496-1 does not allow paramters after parent class name  parent\_class: 'extends' NAME '(' ','.value+ ')'  body: stmt\*  stmt:  | elementary\_data\_type  | non\_parsable\_variable  | assignment\_stmt  | object\_instantiation  | increment\_stmt  | if\_stmt  | switch\_stmt  | for\_stmt  | do\_stmt  | while\_stmt  | comment\_cpp  #TODO: See how to do any character up to newline  comment\_cpp: '//' (NAME | 'floor' | 'class' | 'if' | 'else' | 'for' | 'extends' | NUMBER | '==' | '=' | '{' | ';' | ',' | '-' | '/' | ':' | '?' )\*  # Rule E.1 and A.1  elementary\_data\_type: template? aligned? const? type length NAME array\_length? assigned\_value? ';'  non\_parsable\_variable: template? const? type NAME array\_length? assigned\_value? ';'  #NOTE: Not in 14496-1 but used in 14496-12  template: 'template'  const: 'const'  length: '(' (NUMBER | NAME) ')'  #NOTE: array initialisation with {val1, val2, ...} not in 14496-1  assigned\_value: '=' (value | array\_initialisation)  object\_instantiation: NAME NAME ( '(' ','.value+ ')' )\* array\_length? ';'  variable\_assignment: NAME assigned\_value  assignment\_stmt: variable\_assignment ';'  #TODO: This rule should not allow whitepaces between name and '+'s  variable\_incr: NAME '+' '+'  increment\_stmt: variable\_incr ';'  #NOTE: STRING literal e.g. 'uuid' is not allowed in 14496-1  #NOTE: NUMBER catches decimal, octal, hexadecimal, binary, foating point (scientific noation) and even imaginary number. Too broad for SDL.  value: function | expr | '-'? NUMBER | NAME | STRING  expr: (value operator value) | ( '(' value operator value ')' )  #NOTE: & and && not in 14496-1 but used in 14496-12  operator: operator\_test | operator\_logical | operator\_bin | operator\_math  operator\_math: '+' | '-' | '/' | '\*'  operator\_test: '==' | '<=' | '<' | '>=' | '>' | '!='  operator\_bin: '&' | '|'  operator\_logical: '&' '&' | '|' '|'  function: function\_name '(' value ')'  #NOTE: Only lengthof in 14496-1, floor is used in 14496-12 without definition  function\_name: 'floor' | 'lengthof'  array\_initialisation: '{' ','.value+ '}'  # Rule FC.1  if\_stmt: 'if' '(' condition ')' '{' body '}' else\_if\_stmt? else\_stmt?  else\_if\_stmt: 'else' 'if' '(' condition ')' '{' body '}'  else\_stmt: 'else' '{' body '}'  condition: value  # Rule FC.2  switch\_stmt: 'switch' '(' condition ')' '{' (switch\_case switch\_break?)\* switch\_default? switch\_break?'}'  switch\_break: 'break' ';'  switch\_case: 'case' (NUMBER | NAME | STRING) ':' body?  switch\_default: 'default' ':' body?  # Rule FC.3  for\_stmt: 'for' '(' expression1 ';' expression2 ';' expression3 ')' '{' body '}'  for\_variable\_declaration\_assignment: type NAME array\_length? assigned\_value  expression1:  | variable\_assignment  | for\_variable\_declaration\_assignment  expression2: value  expression3: variable\_incr  # Rule FC.4  do\_stmt: 'do' '{' body '}' 'while' '(' condition ')' ';'  # Rule FC.5  while\_stmt: 'while' '(' condition ')' '{' body '}' |

## References

1. Pegen documentation, <https://we-like-parsers.github.io/pegen/>

# On the keyword template

The template keywork in defined in ISO/IEC 14496-12 and does not belong to the original nor currently developped SDL.

The goal of the keyword is to allow other possible values than the one defined for the field by the assignment operator.

EXAMPLE ⎯

template int(32) rate = 0x00010000; // typically 1.0

In this example, the field rate shall be “0x00010000” for a file complying to this specification. But the keyword template allows a derivate spec to define another value.

From the point of a view of a file parser that only knows about the current specification, it shall throw an error is rate as a different value than “0x00010000”.

The ongoing 8ed of ISOBMFF is attempting to clarify the definition and the reader/writer behaviour.

It should be studied whether template should remain an ISOBMFF extension of the SDL or become a feature of the SDL.

# On optional class member

## Use case #1: A container box

In this scenario, a box is meant to be a container for other boxes. For example, the movie box is defined as follows:

**Movie box**

**Definition**

Box Type: 'moov'  
Container: File  
Mandatory: Yes  
Quantity: Exactly one

The structure-data for a presentation is stored in the single MovieBox which occurs at the top-level of a file. Normally this box is close to the beginning or end of the file, though this is not required.

**Syntax**

aligned(8) class MovieBox extends Box('moov')  
{  
}

From the SDL declaration of the movie box, nothing is specifying what the box can contain. The semantic is also silent on what it contains. Instead, the philosophy of the ISOBMFF specification is to specify where a box can be located and not what it can contain.

Therefore, strictly speaking, the movie box as declared is an empty box.

Another example of a container box is the MetaBox. In this case, specification declare an array of element of the class Box.

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 …  
 Box other\_boxes[]; // optional  
}

Note that the comment mentions optional which would imply that the array may be empty which is currently a topic of discussion in the File Format group (regarding the element track\_IDs[])

More generally, this syntax of declaring a generic class would be possible is the class Box would be an abstract class and that all derive classes would use the SDL mechanism of extension ID. However, those conditions are not met the current ISOBMFF specification.

## Use case #2: Optional box

Another typical case is when a box declares several fields followed some optional boxes. The meta box is also an example of that:

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 HandlerBox(handler\_type) theHandler; // optional  
 PrimaryItemBox primary\_resource[0..1]; // optional  
 DataInformationBox file\_locations; // optional  
 ItemLocationBox item\_locations; // optional  
 ItemProtectionBox protections; // optional  
 ItemInfoBox item\_infos; // optional  
 IPMPControlBox IPMP\_control; // optional  
 ItemReferenceBox item\_refs; // optional  
 ItemPropertiesBox item\_properties; // optional  
 ItemDataBox item\_data; // optional  
 GroupsListBox entity\_groups; // optional  
 Box other\_boxes[]; // optional  
}

## Discussion

## On optional boxes

We assert that it is possible to declare optional boxes in the ISOBMFF because:

1. Boxes can be disambiguated using the box\_type field in the header.
2. The box size in the box header allows the parser to determine if the end of the box is reached. If not, then this means that some additional boxes are present.

## On array of boxes

There seems to be three cases of container box:

1. A box can contain any other boxes.
2. A box can contain zero or more boxes from a any boxes in any order.
3. A box can contain zero or more boxes from a predetermined list of boxes in a specific order.

For case 1), the SDL has some provision but requires using specific tools which limits the freedom of the SDL declaration author.

For case 2), the SDL does not seem to have any tools for that.

For case 3), this is almost enabled in SDL as illustrated in the meta box. The only missing feature is to declare that the boxes are optional.

## Possible solutions

## Feature #1: Optional class members

### General

In SDL, it can happen that a class may contain a given class based on the context it is in. In some environment, this class will be needed in some other this class would not be needed. As a result, the SDL author may want to declare a nested class instance to be optional.

For example:

class Foo  
{  
 unsigned int(8) a;  
 optional Bar bar;  
}

class Log

{

Foo foo;

}

In this class MyClass, the body of Foo does declare bar and but if bar is not in the parsed data, then the member foo.bar does not exist and thus would end up in an unspecified behaviour.

### Optional keyword

## Classes

Classes are the mechanism with which declarations of composite types is performed. Their syntax is as follows.

Rule C.1: Classes

[aligned[(modifier)]] class class\_identifier {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause **Error! Reference source not found.**.

The different *element* entries within the braces (“{“ and “}”) are the definitions of the contained elementary data types (as defined in clause **Error! Reference source not found.**), composite data types (as defined in clause **Error! Reference source not found.**) or syntactic flow control elements (as defined in clause **Error! Reference source not found.**). Furthermore, a particular member element declared in a class may be accessed (considering the scoping rules defined in **Error! Reference source not found.**) using the class member access operator (“**.**”).

Classes may also be encapsulated within other classes. In this case, the element in Rule C.1 is a class itself.

The order of declaration of the *elements* is the same order in which the elements appear in the bitstream.

NOTE As alignment is performed by advancing the current position in the bitstream to be before the next encoded value, it is somewhat redundant but not illegal to declare a class as aligned, and to declare the first element of the class as aligned. In this case the alignment before the first element’s encoded value in the bitstream would be performed using the largest of the two alignment declarations.

A class declaration shall not recursively reference itself.

The next rule describes the use of a declared class.

Rule C.2: Class data types

[optional] class\_identifier *class\_variable\_*identifier**;**

The keyword optional indicates that the class instance may or may not be present. To use this keyword, the class shall be defined with a class identifier.

[Editor’s note: member class mark as typeid? Or should we say that class shall be a derived class of base class for which the class identifier is defined].

Accessing an optional member of class that was not parsed, and thus does not exist in the class instance, result in an unspecified behaviour. Consequently, the value of this class are also unspecified.

NOTE The SDL specification does not provide a mean to check whether the member of class exists.

## Feature #2: Array of boxes for ISOBMFF

### General

In ISOBMFF, there are several places with a box can contains any boxes.

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 …  
 Box other\_boxes[]; // optional  
}

The statement “Box other\_boxes[]” is actually possible in SDL but ISOBMFF doesn’t currently use the SDL polymorphism feature (class identifier in the class declaration) to use this feature.

### Possible solutions

#### New keyword “class id”

Declaring a class identifier keyword to declare which class member in the body should be used as class identifier.

Possible keywords are:

* **typeid**
* **id**
* **class\_id**

EXAMPLE 1 ⎯

class Foo : bit(2) id = 0 {

// note that as "id" is declared it is accessible within this class

// as a constant value and lengthof(id) will return 2

int(5) a; // this a is preceded by the 2 bits of id

}

class Bar {  
 id bit(2) class\_type = 0;  
 int(5) a; // this a is preceded by the 2 bits of id

}

Those two classes Foo and Bar would be syntactically equivalent.

To extend from it, the regular derivation process can be used but the SDL author has to make sure the same class id is defined in each derived class and at the same bit position in the class.

class Foo1 extends Foo {

id bit(2) class\_type = 1;

int(3) b;

}

class Foo2 extends Foo {  
 id bit(2) class\_type = 2;

int(5) c;

}

#### Extending polymorphism declaration

Currently, the polymorphism feature works by defining a field bit which will be written as the first element of the class.

The issue with this in ISOBMFF, and possibly other specifications, is that the box type is written in the second position after the size of the box.

Therefore, the current polymorphism could be extended to allow for declaring the bit offset from which to start reading the class id.

For example, one way would be to add an optional bit offset, when absent the offset is null.

class FooBox : 32+bit(32) type = 'moov' {

unsigned int(32) size;  
 bit(32) type;  
 if (size==1) {  
 unsigned int(64) largesize;  
 } else if (size==0) {  
 // box extends to end of file  
 }  
 if (boxtype=='uuid') {  
 unsigned int(8) usertype[16] = extended\_type;  
 }

…

}

Alternatively, we could also forbid the redeclaration in the class.

class FooBox : 32+bit(32) type = 'moov' {

unsigned int(32) size;  
 // Here is the bit(32) type field but it is already declared in the class declaration   
 if (size==1) {  
 unsigned int(64) largesize;  
 } else if (size==0) {  
 // box extends to end of file  
 }  
 if (boxtype=='uuid') {  
 unsigned int(8) usertype[16] = extended\_type;  
 }

…

}

## Feature #3: An unordered set of boxes

### General

Here the case is that a box contain a sequence of box wherein each box is optional. However, the order of which the boxes appear is not specified and any order is allowed.

We can call this case an unordered set of boxes where set is understood as the concept of unordered set as in C++:

std::unordered\_set is an associative container that contains a set of unique objects of type Key.

So those solutions provide mean to express this type of data.

In spirit, it is also similar to the XML element all:

From 3.8.4.1.3 All-groups , <https://www.w3.org/TR/xmlschema11-1/>

[Definition:]  A **grouping** of a sequence is a set of sub-sequences, some or all of which may be empty, such that each member of the original sequence appears once and only once in one of the sub-sequences and all members of all sub-sequences are in the original sequence.

### Possible solutions

#### New syntax for “an unordered set / all XML element”

Let’s take this box as example and assume the order of the clap and pasp boxes can be any.

class VisualSampleEntry(codingname) extends SampleEntry (codingname)  
{  
 unsigned int(16) pre\_defined = 0;  
..   
 CleanApertureBox clap; // optional  
 PixelAspectRatioBox pasp; // optional  
}

One way of expressing the unordered set it.

class UnorderedBox {

all {

   Second b;

    First a;

  }

}

One parsing can lead to:

First

Second

Another parsing can lead to:

Second

First

To be more explicit, the syntax could also incorporate the way to disambiguate the set, for example:

class UnorderedBox {

all : bit(32) id {

   Second b;

    First a;

  }

}

This mean that the parser has to read id as the next 32 bits to identify it correspond to b.id or a.id.

Another alternative is to declare the base class before the unordered set.

class UnorderedBox {

all : Box {

   FooBox b;

    BarBox a;

  }

}

In this case, FooBox and BarBox derive from Box and per their definition, the parser knows how to disambiguate between a and b.