Building Babel: freeing multimedia processing and delivery from hard-coded formats

Joseph Alfred Thomas-Kerr
University of Wollongong


This paper is posted at Research Online.
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
Building Babel: Freeing multimedia processing and delivery from hard-coded formats

A thesis submitted in fulfilment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Joseph Alfred Ian Thomas-Kerr
Bachelor of Engineering (Honours Class I)
University of Wollongong, 2003

SCHOOL OF ELECTRICAL, COMPUTER AND TELECOMMUNICATIONS ENGINEERING
2009
Abstract

The amount of multimedia content available via the Internet, and the number of formats in which it is encoded, stored and delivered continues to grow rapidly. So too the number and diversity of the devices and software applications which produce, process and consume such content. This constantly changing landscape presents an increasing challenge to interoperability, since more and more software and hardware must be upgraded as new formats are developed. However, many of the operations performed on multimedia content are similar across coding formats. In recognising this, this thesis proposes several approaches to format-independent media processing, with an emphasis on content delivery. This considerably simplifies interoperability, since support for a new content format may be provided by disseminating a data file, rather than requiring application and device providers to extend and modify their software and hardware.

A fundamental requirement for format-independence is the ability to describe the structure of any given format in a way that exposes how it may be fragmented for delivery or processing, and how other data important to the processing (for instance temporal or scalability parameters) can be extracted from the binary data. Several
meta-syntax languages are evaluated that (to greater or lesser degree) perform this function. Of these, the most suitable for general use in format-independent processors is found to be MPEG-21’s Bitstream Syntax Description Language (BSDL). Its general suitability notwithstanding, BSDL exhibits several critical flaws when used to describe and process modern content formats. In response, this thesis proposes several new features for the language which significantly reduce processing complexity, and provide extensibility for complex data types. These features are implemented and validated using bitstreams of real-world length, which enable a linear response of approximately 10 times the speed of playback (on the particular test machine used), for videos up to one hour in duration.

Digital media increasingly encompasses a wide range of metadata, as well as collections of related content (a DVD and it’s “special features”, for instance). Several recent standards address generic virtual containers for such rich content. While these standards—which include MPEG-21 and TVAnytime—provide numerous tools for interacting with rich media objects, they do not provide a framework for streaming or delivery of such data. This thesis presents the Bitstream Binding Language (BBL), a format-independent tool that describes how multimedia content and metadata may be bound into delivery formats. Using a BBL description, a generic processor can map rich content (an MPEG-21 Digital Item, for example) into a streaming or static delivery format. BBL provides a universal syntax for fragmentation and packetisation of both XML and binary data, and allows new content and metadata formats to be delivered without requiring the addition of new software to the delivery infras-
Abstract

structure. The BBL framework is validated and tested against a number of application scenarios including a format-independent streaming server, generic metadata syntax translation, virtual container assembly, and a format-independent hinter.

Finally, it is observed that much of the semantic metadata that is generated to describe multimedia content could also be used to improve the decisions that must be made in order to transmit it effectively. Indeed, methods have been proposed for using *specific* semantic concepts in the delivery process. However, until now, no high-level system has been proposed that is able to take arbitrary semantic metadata, and utilise it in the multimedia delivery decision-making process. This thesis proposes such a system. It combines the aforementioned semantic concepts with other existing work in Rate-Distortion Optimisation for multimedia delivery, scalable content formats, and syntax description, and then develops a generalised framework to permit an arbitrary range of semantic metadata and optimisation techniques to be utilised. This objective is accomplished by utilising *schema* languages to describe the details of any given content or metadata, so that declarative mapping rules can be specified for translating from format-specific data points to format-independent concepts that are directly used by the framework. This translation can then be performed using software or hardware that knows nothing about the specific format it is processing.

This thesis describes a particular embodiment of the semantic-aware multimedia delivery system which was implemented in order to verify its key assertions. It presents the results of subjective testing that was performed on several short news clips encoded using H.264/SVC scalable video coding, and Scalable-To-Lossless (SLS) au-
dio coding. Each clip was adapted to four target bitrates, using both of two methods:

(a) using the semantic-aware system to devote a greater proportion of the available bandwidth to that part of the content (audio or video) that was conveying more of the semantics at any given time; and

(b) at a constant bit-rate with the same average rate as clip (a).

Test participants were shown each pair of clips (a and b) in a random order and were asked to evaluate which was more successful at conveying the meaning of the story. The result of this subjective testing was a 72% preference for those clips which had been adapted so as to devote more bandwidth to the semantically-important parts of the content.
Contents

1 Building Babel: Constructing a framework to remove language-dependence from multimedia communication 1

1.1 A different frame of reference for multimedia technology . . . . . . . . . 3
1.2 Many layers of abstraction . . . . . . . . . . . . . . . . . . . . . . 6
1.3 Describing the structure of each abstraction . . . . . . . . . . . . . . 8
1.4 Implications of the model . . . . . . . . . . . . . . . . . . . . . . . 9
  1.4.1 Surveying the site . . . . . . . . . . . . . . . . . . . . . . . 10
  1.4.2 Laying the foundations . . . . . . . . . . . . . . . . . . . . 10
  1.4.3 Babel’s formwork . . . . . . . . . . . . . . . . . . . . . . . 10
  1.4.4 Erecting the tower . . . . . . . . . . . . . . . . . . . . . . . 11
  1.4.5 What are you trying to say? . . . . . . . . . . . . . . . . . 12
  1.4.6 Scattered to the winds . . . . . . . . . . . . . . . . . . . . . 12
  1.4.7 Tearing it all down again . . . . . . . . . . . . . . . . . . . 12
1.5 Summary of contributions in this work . . . . . . . . . . . . . . . . 13
1.6 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  1.6.1 Journal Articles . . . . . . . . . . . . . . . . . . . . . . . . 16
  1.6.2 Conference Papers . . . . . . . . . . . . . . . . . . . . . . . 16
  1.6.3 MPEG Standards . . . . . . . . . . . . . . . . . . . . . . . . 18
  1.6.4 Patents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2 Surveying the site: A review of the literature

2.1 Syntactic Metalanguages

2.1.1 Bitstream Syntax Description Language

2.1.2 Flavor/XFlavor

2.1.3 BFlavor

2.1.4 Comparison between BSDL and XFlavor

2.2 Media and Metadata Fragmentation

2.2.1 Extensible Stylesheet Language Transformations

2.2.2 XML Path Language

2.3 Format-independent Media Delivery

2.3.1 Streaming Server Architectures

2.3.2 Metadata Streaming

2.3.3 Content and Metadata packages

2.3.4 Summary

2.4 Scalable Content and Semantics

2.4.1 Scalable Content Formats

2.4.2 Content Adaptation

2.4.3 Rate-Distortion Optimised Packet Scheduling

2.4.4 Adaptive Delivery using Semantics

2.4.5 Ontology-based Metadata

2.4.6 Summary

2.5 Conclusion

3 Laying the foundations: A universal metasyntax for multimedia

3.1 A brief tutorial on BSDL

3.2 An analysis of BSDL with recent coding formats
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Entropy-coded header fields</td>
<td>72</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Parameter Sets</td>
<td>74</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Dynamic length bit-fields</td>
<td>79</td>
</tr>
<tr>
<td>3.3</td>
<td>Extensions to BSDL to allow its use with recent content formats</td>
<td>81</td>
</tr>
<tr>
<td>3.3.1</td>
<td>ECMAScript implementation of extension datatypes</td>
<td>81</td>
</tr>
<tr>
<td>3.3.2</td>
<td>XPath Variable declarations</td>
<td>86</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Dynamic bit-length simple types</td>
<td>88</td>
</tr>
<tr>
<td>3.4</td>
<td>Results</td>
<td>90</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusions</td>
<td>92</td>
</tr>
<tr>
<td>4</td>
<td>Babel’s formwork: Format-independent streaming and delivery</td>
<td>95</td>
</tr>
<tr>
<td>4.1</td>
<td>Developing A Framework For Format-Independent Multimedia Delivery</td>
<td>98</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Application Scenarios</td>
<td>98</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Requirements</td>
<td>101</td>
</tr>
<tr>
<td>4.1.3</td>
<td>A model for generic delivery of rich media content</td>
<td>103</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementing The Model: The Bitstream Binding Language</td>
<td>107</td>
</tr>
<tr>
<td>4.2.1</td>
<td>BBL Server Architecture</td>
<td>112</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Configuration</td>
<td>114</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Binary Abstraction</td>
<td>116</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Fragmentation</td>
<td>117</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Packetisation</td>
<td>118</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Post Processing</td>
<td>120</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Output Mapping</td>
<td>120</td>
</tr>
<tr>
<td>4.3</td>
<td>Evaluating The Bitstream Binding Language</td>
<td>120</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Pre-processing of content</td>
<td>122</td>
</tr>
</tbody>
</table>
4.3.2 Pre-compilation of BBL Instructions .................................. 123
4.4 Conclusion ........................................................................... 124

5 Erecting the tower: Applications for format-independent streaming and delivery 127

5.1 A format-independent streaming server .................................. 127
  5.1.1 A model for format-independent streaming .......................... 129
  5.1.2 BBL For H.264/AVC over RTP ......................................... 132
  5.1.3 Results ........................................................................... 134
5.2 Generic multimedia syntax translation ................................. 136
  5.2.1 An example generic syntax translation ............................... 138
5.3 Virtual container assembly .................................................... 144
  5.3.1 ISO/Quicktime File Output Handler ................................. 145
  5.3.2 Example—Scalable Generic Streaming of H.264/AVC over RTP .................................................. 149
5.4 Conclusion ........................................................................... 150

6 What are you trying to say?: Format-independent semantic-aware streaming and delivery 153

6.1 Semantics in the delivery process .......................................... 155
6.2 Features .............................................................................. 158
  6.2.1 Format-independence ...................................................... 158
  6.2.2 Semantic-independence ................................................... 159
  6.2.3 Multiple optimisation algorithms ..................................... 160
  6.2.4 Segmentation and association .......................................... 160
6.3 A framework for format-independent semantic-aware multimedia delivery ................................................. 161
  6.3.1 Delivery Node .................................................................. 162
6.3.2 Semantic R-D hinter ........................................ 165
6.3.3 An Ontology for Semantic Distortion .................. 180
6.3.4 Summary ..................................................... 188
6.4 Subjective Testing ............................................. 190
  6.4.1 Methodology ............................................. 190
  6.4.2 Results .................................................. 198
6.5 Conclusion .................................................... 199

7 Scattered to the winds: Conclusions and Future work 201
  7.1 Future Work ................................................ 207

Bibliography ....................................................... 211

A Tearing it all down again: Toward format-independent decoding A–1
  A.1 Objectives .................................................. A–4
  A.2 Usage scenarios for reconfigurable coding .............. A–5
  A.3 An RMC bitstream .......................................... A–7
  A.4 The CAL Actor Language .................................. A–8
    A.4.1 Dataflow oriented processing ......................... A–9
    A.4.2 Hierarchical modular design ......................... A–11
    A.4.3 Communication protocols ............................. A–11
    A.4.4 Nondeterministic scheduling and explicit parallelism A–12
    A.4.5 Summary: CAL ....................................... A–12
  A.5 The Reconfigurable Media Coding framework .......... A–13
    A.5.1 Decoder Description Language ....................... A–14
  A.6 Bitstream Syntax Description ............................. A–15
  A.7 Parser Generation .......................................... A–20
A.7.1 Preprocessing                      A–21
A.7.2 Finite State Machine              A–22
A.7.3 Field bit-length                  A–30
A.7.4 CAL Templates                     A–32
A.8 Conclusion                          A–33
A.8.1 Future Work                      A–34

B  Data tables for BSDL                  B–1

C  Bitstream Binding Language Specification         C–1

D  Bitstream Binding Language XML Schema         D–1

E  Listings for multi-channel rich media delivery E–1

F  Listings for generic syntax translation      F–1

G  Listings for virtual container assembly     G–1

H  Listings for H.264/AVC Streaming and delivery H–1

I  Listings for Intelligent Media Delivery     I–1

J  Listings for Reconfigurable Media Coding    J–1
Statement of Originality

This is to certify that the work described in this thesis is entirely my own, except where due reference is made in the text.

No work in this thesis has been submitted for a degree to any other university or institution.

Signed

Joseph Alfred Ian Thomas-Kerr

12th of May, 2009
Acknowledgments

I am indebted to many people, without whom this thesis would not be. To them I offer my heartfelt gratitude.

Firstly, to my Creator, the source of all creativity and inspiration, and my closest companion.

To my wife, Michelle, my lover, best friend, confidant, and editor.

To my supervisors, Ian Burnett and Christian Ritz, whose guidance and reassurance were indispensable.

Campbell Thomas, my father-in-law, for the affirmation which helped me persevere through trying circumstances.

And to Molly and Edward, heartbeats at my feet, never leaving, always believing.
List of Figures

1.1 The diversity and complexity of multimedia delivery has grown exponentially ........................................ 2

1.2 There are many levels of abstraction for a single piece of content ....................................................... 5

2.1 Syntactic Metalanguages ................................................................. 20

2.2 These examples show a small part of the MPEG-4 Video syntax in several of the numerous syntax description metalanguages. .................................................. 23

2.3 Media and Metadata Fragmentation .................................................. 28

2.4 Format-independent Media Delivery ................................................. 30

2.5 Monolithic Streaming Server .......................................................... 31

2.6 Hinted Streaming Server ................................................................. 33

2.7 gBSD-Based streaming server ......................................................... 34

2.8 Metadata Streaming .................................................................. 35

2.9 Annodex media format [67] ............................................................. 38

2.10 Content/Metadata packages .......................................................... 39

2.11 Fundamental boxes in an ISO/Quicktime file ................................. 40

2.12 A Music Player Multimedia Application Format combines technologies from MPEG-1, -4, -7, -21 and JPEG to provide an augmented music container .......................................................... 42

2.13 Scalable Content Formats .............................................................. 44

2.14 Content Adaptation ................................................................. 47
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.15</td>
<td>Dynamic Adaptation using MPEG-21 (adapted from [94])</td>
</tr>
<tr>
<td>2.16</td>
<td>Bandwidth-efficient Dynamic Adaptation using MPEG-21 (adapted from [94])</td>
</tr>
<tr>
<td>2.17</td>
<td>Rate-Distortion Optimised Packet Scheduling</td>
</tr>
<tr>
<td>2.18</td>
<td>Adaptive Delivery using Semantics</td>
</tr>
<tr>
<td>2.19</td>
<td>Freeing multimedia from hard-coded formats across the delivery chain</td>
</tr>
<tr>
<td>3.1</td>
<td>The Bitstream Syntax Description Language is finding use at multiple points in the multimedia delivery chain</td>
</tr>
<tr>
<td>3.2</td>
<td>A H.264/AVC bitstream contains numerous long-term references</td>
</tr>
<tr>
<td>3.3</td>
<td>BSDL Schemata and test results demonstrating quadratic-time performance of the SPS dereference operation</td>
</tr>
<tr>
<td>3.4</td>
<td>Average Processing Time to generate a BSDL description of H.264/AVC bitstreams of varying length, by proposed modification</td>
</tr>
<tr>
<td>3.5</td>
<td>Average Processing Time to generate a BSDL description of H.264/AVC bitstreams of varying length, all modifications</td>
</tr>
<tr>
<td>4.1</td>
<td>There is a many-to-many relationship between the formats in which content is available and the channels across which users wish to consume them</td>
</tr>
<tr>
<td>4.2</td>
<td>MPEG Multimedia Application Formats are one example where generic syntax translation can be used to enhance interoperability</td>
</tr>
<tr>
<td>4.3</td>
<td>A binding language could be used to assemble virtual containers</td>
</tr>
<tr>
<td>4.4</td>
<td>A model for multimedia content delivery</td>
</tr>
<tr>
<td>4.5</td>
<td>The System architecture of the prototype BBL server</td>
</tr>
<tr>
<td>5.1</td>
<td>BBL-based streaming server</td>
</tr>
<tr>
<td>5.2</td>
<td>A model of multimedia streaming</td>
</tr>
<tr>
<td>5.3</td>
<td>CPU and Memory utilisation for prototype BBL processor with H.264/AVC over RTP</td>
</tr>
<tr>
<td>5.4</td>
<td>BBL lets users consume content in a format their device can understand</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.5</td>
<td>Files involved in translating MAF files to MP3/ID3 v2</td>
</tr>
<tr>
<td>5.6</td>
<td>Fundamental boxes in an ISO/Quicktime file</td>
</tr>
<tr>
<td>5.7</td>
<td>Static and global characteristics are specified by a global parameter set. Parameters specific to a packet of media data are specified with that packet</td>
</tr>
<tr>
<td>5.8</td>
<td>The structure of the ISO/Quicktime file handler global parameters</td>
</tr>
<tr>
<td>5.9</td>
<td>The packet-level parameters used by the ISO/Quicktime File handler</td>
</tr>
<tr>
<td>6.1</td>
<td>A framework for semantic-aware multimedia delivery</td>
</tr>
<tr>
<td>6.2</td>
<td>Coupling multimedia semantics and a delivery optimisation process requires numerous disparate technologies</td>
</tr>
<tr>
<td>6.3</td>
<td>News reports have a regular structure</td>
</tr>
<tr>
<td>6.4</td>
<td>An architecture for Multimedia Delivery that incorporates content semantics</td>
</tr>
<tr>
<td>6.5</td>
<td>A delivery node uses content hints to perform R-D optimisation</td>
</tr>
<tr>
<td>6.6</td>
<td>The semantic hinter computes R-D metadata based on content syntax and semantics</td>
</tr>
<tr>
<td>6.7</td>
<td>A binary schema can be used to expose the structure and data in a binary bitstream as OWL/RDF</td>
</tr>
<tr>
<td>6.8</td>
<td>Mapping rules can be used to translate from format-specific structures into the format-independent metadata needed for a semantic-aware RDO delivery framework</td>
</tr>
<tr>
<td>6.9</td>
<td>An ontology for Semantic Distortion</td>
</tr>
<tr>
<td>6.10</td>
<td>Example of the semantic annotation of an Audio-Visual Clip</td>
</tr>
<tr>
<td>6.11</td>
<td>Example instances of Semantic Distortion classes describing the Data Units of a H.264/AVC bitstream</td>
</tr>
<tr>
<td>6.12</td>
<td>Example instances of Semantic Distortion classes describing the distortion of a H.264/AVC bitstream</td>
</tr>
<tr>
<td>6.13</td>
<td>The test corpus consists of four short video clips that were both semantically-adapted and bitrate-adapted to various target rates</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>6.14</td>
<td>Semantic adaptation diverts more bits to the portion of the content containing more of the <em>meaning</em></td>
</tr>
<tr>
<td>6.15</td>
<td>Subjective testing shows a 72% preference for Semantically-aware multimedia delivery</td>
</tr>
<tr>
<td>7.1</td>
<td>The diversity and complexity of the multimedia landscape is growing exponentially (reprised from Figure 4.1)</td>
</tr>
<tr>
<td>A.1</td>
<td>A general view of an RMC bitstream</td>
</tr>
<tr>
<td>A.2</td>
<td>CAL blocks can be synthesised into numerous implementations</td>
</tr>
<tr>
<td>A.3</td>
<td>Reconfigurable Media Coding implementation model</td>
</tr>
<tr>
<td>A.4</td>
<td>Components of the parser generation process</td>
</tr>
<tr>
<td>A.5</td>
<td>FSM fragment for a choice particle</td>
</tr>
<tr>
<td>A.6</td>
<td>Reconfigurable Media Coding has the potential to drastically shorten the time required to deploy new multimedia technology</td>
</tr>
</tbody>
</table>
List of Tables

3.1 BSDL-defined ECMAScript functions .............................. 85

4.1 BRL Fragmentation Rules ................................. 118

4.2 On-The-Fly Bbl Processing, Performance Results .............. 121

5.1 CPU and Memory utilisation for prototype BBL processor with H.264/AVC over RTP .................................. 135

A.1 Extract from MPEG-4 syntax specification ....................... A–17

B.1 Data table for Figure 3.4 ................................ B–2

B.2 Data table for Figure 3.5 ................................ B–3

I.1 Instructions given to subjective testing candidates for the experiment described in Chapter 6, Section 6.4 ....................... I–16
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A BSDL Description exposes the fields in a multimedia bitstream, in this case a H.264/AVC video</td>
</tr>
<tr>
<td>3.2</td>
<td>A BSDL Schema describes the structure common to all H.264/AVC video bitstreams</td>
</tr>
<tr>
<td>3.3</td>
<td>Sequence Parameter Set dereferencing-XPath expression 1</td>
</tr>
<tr>
<td>3.4</td>
<td>Sequence Parameter Set dereferencing-XPath expression 2</td>
</tr>
<tr>
<td>3.5</td>
<td>Sequence Parameter Set dereferencing-XPath expression 3</td>
</tr>
<tr>
<td>3.6</td>
<td>BSDL test schema for [last()]</td>
</tr>
<tr>
<td>3.7</td>
<td>BSDL test schema for preceding::</td>
</tr>
<tr>
<td>3.8</td>
<td>BSDL schema fragment showing union data types—excerpt from the MPEG conformance schema for H.264/AVC [124]</td>
</tr>
<tr>
<td>3.9</td>
<td>BSDL Schema for the Exp. Golomb as defined by H.264</td>
</tr>
<tr>
<td>3.10</td>
<td>Sequence Parameter Set dereferencing after proposed amendment-XPath expression</td>
</tr>
<tr>
<td>3.11</td>
<td>A variable declared for the set of SPS</td>
</tr>
<tr>
<td>3.12</td>
<td>BSDL Schema fragment describing the frameNum field of an H.264/AVC bitstream</td>
</tr>
<tr>
<td>4.1</td>
<td>BBL instructions for TV program fragmentation</td>
</tr>
<tr>
<td>4.2</td>
<td>BBL instructions for MPEG-2 video fragmentation</td>
</tr>
<tr>
<td>4.3</td>
<td>Part of BSDL schema for MPEG-2 system streams</td>
</tr>
<tr>
<td>5.1</td>
<td>BBL temporal model</td>
</tr>
<tr>
<td>5.2</td>
<td>BBL Instructions for streaming H.264/AVC content</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.3</td>
<td>MAF BS Schema excerpt</td>
</tr>
<tr>
<td>5.4</td>
<td>MP3/ID3 v2 BS Schema excerpt</td>
</tr>
<tr>
<td>5.5</td>
<td>BBL Instructions for MAF to MP3/ID3</td>
</tr>
<tr>
<td>5.6</td>
<td>Abbreviated BBL instructions for scalable generic streaming of H.264/AVC on RTP</td>
</tr>
<tr>
<td>6.1</td>
<td>Augmented BSDL test schema exposing the ancestor Data Units of an SVC PPS</td>
</tr>
<tr>
<td>6.2</td>
<td>XSLT fragment annotating pps elements with ancestor metadata</td>
</tr>
<tr>
<td>6.3</td>
<td>A SWRL mapping rule stating that a NAL Unit is an RDO Data Unit</td>
</tr>
<tr>
<td>6.4</td>
<td>A not-quite-complete rule for specifying the interdependency between a PPS and the SPS it references</td>
</tr>
<tr>
<td>6.5</td>
<td>A SWRL rule that specifies the Semantic Distortion of English Communication</td>
</tr>
<tr>
<td>6.6</td>
<td>Chou’s R-D optimisation algorithm</td>
</tr>
<tr>
<td>6.7</td>
<td>A semantic rule from the test data asserting that still images have a (relative) SD of 0.5</td>
</tr>
<tr>
<td>A.1</td>
<td>Part of the BSDL Schema for MPEG-4 Video</td>
</tr>
<tr>
<td>A.2(a)</td>
<td>Input schema fragment showing an include directive</td>
</tr>
<tr>
<td>A.2(b)</td>
<td>XSLT fragment showing preprocessor templates</td>
</tr>
<tr>
<td>A.2(c)</td>
<td>Fragment of intermediate result tree composed of base and included schema components</td>
</tr>
<tr>
<td>A.3(a)</td>
<td>Input schema fragment showing a complex type and sequence particles</td>
</tr>
<tr>
<td>A.3(b)</td>
<td>XSLT fragment for the linearise component</td>
</tr>
<tr>
<td>A.3(c)</td>
<td>Fragment of intermediate result tree composed of read instructions</td>
</tr>
<tr>
<td>A.4(a)</td>
<td>Intermediate result tree fragment composed of read instructions</td>
</tr>
<tr>
<td>A.4(b)</td>
<td>XSLT fragment showing FSM transition templates</td>
</tr>
<tr>
<td>A.4(c)</td>
<td>Fragment of CAL output showing FSM schedule</td>
</tr>
<tr>
<td>A.5(a)</td>
<td>Intermediate result tree fragment composed of read instructions</td>
</tr>
</tbody>
</table>
A.5(b) XSLT fragment showing action templates .................................. A–27
A.5(c) Output CAL fragment showing several actions .......................... A–28
A.6(a) Input schema fragment showing an element with simple type ..... A–30
A.6(b) XSLT fragment showing length constant templates .................. A–30
A.6(c) Output CAL fragment showing length constants ....................... A–32
A.7(a) Intermediate result tree .......................................................... A–32
A.7(b) XSLT fragment showing CAL templates for an FSM & transition . A–32
A.7(c) Output fragment showing FSM transition CAL statements ......... A–33

D.1 XML Schema for the Bitstream Binding Language ........................ D–1
E.1 BS Schema for MPEG-2 system streams ..................................... E–1
F.1 BS Schema for ID3v2 ................................................................. F–1
F.2 BS Schema for MPEG Multimedia Application Format ................. F–3
F.3 BBL description of generic syntax translation of ID3v2 metadata into MAF format’ ................................................................. F–13
G.1 MPEG File Format Handler Parameters ...................................... G–1
H.1 BS Schema describing H.264/AVC ............................................... H–1
H.2 BBL binding describing H.264/AVC content delivery over RTP .... H–11
H.3 BBL binding describing H.264/AVC content delivery via an ISO file package ................................................................. H–13
I.1 XML Schema for RDO metadata ................................................. I–1
I.2 BSDL Schema for MPEG SLS Audio ........................................... I–2
I.3 XSLT Stylesheet to annotate SLS BSDL data with RDO metadata ... I–8
I.4 SWRL rules to generate Semantic Distortion based on select CYC features ................................................................. I–9


**LIST OF LISTINGS**

| J.1 | BS Schema describing MPEG-4 Visual Simple Profile | J–1 |
| J.2 | BS Schema describing MPEG-4 Visual Simple Profile (simple types) | J–22 |
| J.3 | XSLT stylesheet for parser generation (main) | J–28 |
| J.4 | XSLT stylesheet for parser generation (preprocess) | J–31 |
| J.5 | XSLT stylesheet for parser generation (remove redundant sequences) | J–31 |
| J.6 | XSLT stylesheet for parser generation (inline complex extension/restrictions) | J–31 |
| J.7 | XSLT stylesheet for parser generation (add names) | J–32 |
| J.8 | XSLT stylesheet for parser generation (resolve Includes/Imports) | J–33 |
| J.9 | XSLT stylesheet for parser generation (functions) | J–34 |
| J.10 | XSLT stylesheet for parser generation (globals) | J–39 |
| J.11 | XSLT stylesheet for parser generation (length) | J–41 |
| J.12 | XSLT stylesheet for parser generation (linearisation) | J–44 |
| J.13 | XSLT stylesheet for parser generation (FSM) | J–52 |
| J.14 | XSLT stylesheet for parser generation (Actions) | J–54 |
| J.15 | XSLT stylesheet for parser generation (Priorities) | J–62 |
| J.16 | XSLT stylesheet for parser generation (CAL templates) | J–63 |
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>3rd Generation Partnership Project</td>
</tr>
<tr>
<td>AAC</td>
<td>Advanced Audio Coding</td>
</tr>
<tr>
<td>ADIF</td>
<td>Audio Data Interchange Format</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APIC</td>
<td>Associated Picture</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASN.1</td>
<td>Abstract Syntax Notation 1</td>
</tr>
<tr>
<td>AU</td>
<td>Access Unit</td>
</tr>
<tr>
<td>AVC</td>
<td>Advanced Video Coding</td>
</tr>
<tr>
<td>AVI</td>
<td>Audio-Video Interchange</td>
</tr>
<tr>
<td>BBL</td>
<td>Bitstream Binding Language</td>
</tr>
<tr>
<td>BF\text{Flavor}</td>
<td>BSDL Flavor</td>
</tr>
<tr>
<td>BintoBSD</td>
<td>Binary to BSD</td>
</tr>
<tr>
<td>BPath</td>
<td>Binary Path Language</td>
</tr>
<tr>
<td>BS Schema</td>
<td>Bitstream Syntax Schema</td>
</tr>
<tr>
<td>BSD</td>
<td>Bitstream Syntax Description</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>BSDL</td>
<td>Bitstream Syntax Description Language</td>
</tr>
<tr>
<td>BSDtoBin</td>
<td>BSD to Binary</td>
</tr>
<tr>
<td>CAL</td>
<td>CAL Actor Language</td>
</tr>
<tr>
<td>CALML</td>
<td>CAL XML Syntax</td>
</tr>
<tr>
<td>CDATA</td>
<td>Character Data</td>
</tr>
<tr>
<td>CGS</td>
<td>Coarse-Grained Scalability</td>
</tr>
<tr>
<td>CMML</td>
<td>Continuous Media Markup Language</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-Static Data Flow</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>DAB</td>
<td>Digital Audio Broadcasting</td>
</tr>
<tr>
<td>DAML</td>
<td>DARPA Agent Markup Language</td>
</tr>
<tr>
<td>DARPA</td>
<td>Defense Advanced Research Projects Agency</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DDL</td>
<td>Decoder Description Language</td>
</tr>
<tr>
<td>DI</td>
<td>Digital Item</td>
</tr>
<tr>
<td>DIDL</td>
<td>Digital Item Description Language</td>
</tr>
<tr>
<td>DMB</td>
<td>Digital Multimedia Broadcasting</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcasting</td>
</tr>
<tr>
<td>DVD</td>
<td>Digital Versatile Disc</td>
</tr>
<tr>
<td>EBNF</td>
<td>Enhanced Backus Naur Form</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ECMA</td>
<td>European Computer Manufacturers Association</td>
</tr>
<tr>
<td>EXIF</td>
<td>EXchangeable Image File format</td>
</tr>
<tr>
<td>FGS</td>
<td>Fine-Granular Scalability</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FLAVOR</td>
<td>Formal Language for Audio Visual Object Representation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GIF</td>
<td>Graphics Interchange Format</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix</td>
</tr>
<tr>
<td>GZIP</td>
<td>GNU Zip</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>ID</td>
<td>Identification</td>
</tr>
<tr>
<td>ID3</td>
<td>IDentify mp3</td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse Discrete Cosine Transform</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organisation</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LQP</td>
<td>Quantisation parameter from the SVC reference software</td>
</tr>
<tr>
<td>MAF</td>
<td>Multimedia Application Format</td>
</tr>
<tr>
<td>MB</td>
<td>Macroblock</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>MDR-V500</td>
<td>A model of stereo headphones made by Sony</td>
</tr>
<tr>
<td>MP3</td>
<td>MPEG Layer 3 audio format</td>
</tr>
<tr>
<td>MPEG</td>
<td>Motion Picture Experts Group</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>NAL</td>
<td>Network Abstraction Layer</td>
</tr>
<tr>
<td>NALU</td>
<td>NAL Unit</td>
</tr>
<tr>
<td>NPT</td>
<td>Normal Playback Time</td>
</tr>
<tr>
<td>OIL</td>
<td>Ontology Inference Layer</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PCM</td>
<td>Pulse-Coded Modulation</td>
</tr>
<tr>
<td>PES</td>
<td>Packetised Elementary Stream</td>
</tr>
<tr>
<td>PPS</td>
<td>Picture Parameter Set</td>
</tr>
<tr>
<td>QCIF</td>
<td>Quarter-Common Interchange Format</td>
</tr>
<tr>
<td>QName</td>
<td>Qualified Name</td>
</tr>
<tr>
<td>QT</td>
<td>QuickTime</td>
</tr>
<tr>
<td>QTFF</td>
<td>QuickTime file format</td>
</tr>
<tr>
<td>QVGA</td>
<td>Quarter VGA</td>
</tr>
<tr>
<td>RDDD</td>
<td>Rights Data Dictionary</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RDFS</td>
<td>RDF Schema</td>
</tr>
<tr>
<td>RDO</td>
<td>Rate-Distortion Optimisation</td>
</tr>
<tr>
<td>RGB</td>
<td>Red-Green-Blue</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>RFC</td>
<td>Request for Comment</td>
</tr>
<tr>
<td>RMC</td>
<td>Reconfigurable Media Coding</td>
</tr>
<tr>
<td>RQP</td>
<td>Quantisation parameter from the SVC reference software</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SBS</td>
<td>Special Broadcasting Service</td>
</tr>
<tr>
<td>SCR</td>
<td>System Clock Reference</td>
</tr>
<tr>
<td>SD</td>
<td>Semantic Distortion</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>SI</td>
<td>Spatial Information</td>
</tr>
<tr>
<td>SLOC</td>
<td>Source Lines of Code</td>
</tr>
<tr>
<td>SLS</td>
<td>Scalable-to-Lossless coding</td>
</tr>
<tr>
<td>SMPTE</td>
<td>Society of Motion-Picture and Television Engineers</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>SPS</td>
<td>Sequence Parameter Set</td>
</tr>
<tr>
<td>SSRC</td>
<td>Synchronization Source</td>
</tr>
<tr>
<td>SVC</td>
<td>Scalable Video Coding</td>
</tr>
<tr>
<td>SWQL</td>
<td>Semantic Web Query Language</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>TI</td>
<td>Temporal Information</td>
</tr>
<tr>
<td>TIFF</td>
<td>Tagged Image File Format</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TS</td>
<td>Transport Stream</td>
</tr>
<tr>
<td>UMA</td>
<td>Universal Multimedia Access</td>
</tr>
<tr>
<td>URI</td>
<td>Universal Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>VC.1</td>
<td>Video Codec 1</td>
</tr>
<tr>
<td>VGA</td>
<td>Video Graphics Array</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High-Speed Integrated Circuit</td>
</tr>
<tr>
<td>VLC</td>
<td>Variable-Length Coding</td>
</tr>
<tr>
<td>VOL</td>
<td>Video Object Layer</td>
</tr>
<tr>
<td>VOP</td>
<td>Video Object Plane</td>
</tr>
<tr>
<td>WBXML</td>
<td>WAP Binary XML</td>
</tr>
<tr>
<td>WAP</td>
<td>Wireless Access Protocol</td>
</tr>
<tr>
<td>XFlavor</td>
<td>XML FLAVOR</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XQuery</td>
<td>XML Query Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Description</td>
</tr>
<tr>
<td>XSL</td>
<td>XML Stylesheet Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>XSL Transformations</td>
</tr>
<tr>
<td>XStream</td>
<td>XML Stream</td>
</tr>
<tr>
<td>YCrCb</td>
<td>Luminance-Chrominance colour space</td>
</tr>
</tbody>
</table>
Chapter 1

Building Babel

Constructing a framework to remove language-dependence from multimedia communication

The MP3\(^1\) digital audio format was first published in 1991. Only in the last five years, however, has Moore’s law\(^2\) allowed such audio to be decoded by battery-powered, portable devices \(^3\), fundamentally changing the way most people obtain and consume music, and making MP3 a household name. In a similar period, MP3 has been joined by a plethora of other multimedia\(^3\) formats that are commonly used by content creators to encode audio and video: Windows Media \(^4\), Quicktime \(^5\), Ogg \(^6\), Flash \(^7\), MPEG-2 \(^8\), and MPEG-4 \(^9\) to name a few. The diversity of the devices used to deliver, process, and consume multimedia content has also increased dramatically \(^10\) (Figure 1.1 on the following page). This proliferation of multimedia formats and devices presents an escalating challenge to interoperability between the format in

\(^{1}\)MPEG (Motion Picture Experts Group) Layer 3 Audio \([1]\)

\(^{2}\)the prediction that the number of transistors on a comparable integrated circuit will double approximately every two years \([2]\).

\(^{3}\)for the purposes of this thesis, Multimedia is taken to mean digitally encoded audio and/or video.
Figure 1.1: The diversity and complexity of multimedia delivery has grown exponentially

which content is produced and the devices on which users wish to consume it [11].

For example: a new video codec is developed, say H.264/AVC (the ITU-T H.264 / MPEG Advanced Video Coding standard) [12]. Before this codec can become widely useful, a critical mass of the manufacturers of content creation hardware and software, of delivery node hardware and software, of content processing hardware and software, and of consumption hardware and software, must all develop additional functionality to support the new codec. Conversely, a new delivery node that is developed must support a growing range of content formats, if it is to provide access to a significant portion of the encoded content available on digital networks.

In general, this leads to an exponential growth in the total amount of effort required to exploit developments in multimedia technology.
This thesis is about ways to deal with this challenge. Rather than define the operations performed on multimedia on the basis of the bit-by-bit semantics which define and distinguish content formats, this work proposes that many such operations (e.g. multimedia delivery) are in fact very similar, regardless of the actual format of the data. Thus, the systems proposed herein are format-independent, and the (nevertheless important!) bit-by-bit details provided as an input to the system, rather than hard-coded into it. Furthermore, Appendix A develops part of a system that transcends a-priori defined media codecs, making content self-describing, and thus able to reconfigure as required to suit content and channel characteristics. At a conceptual level, this thesis is motivated by a frame of reference that views a codec as but one of many possible abstractions for a piece of multimedia content, as described below (Section 1.1).

### 1.1 A different frame of reference for multimedia technology

Much of the effort in multimedia research is focused on one or the other of the media itself, or on metadata that describes the media. There has been a great deal of work on increasingly efficient ways to encode media data, making it smaller and hence cheaper to store and transmit\(^4\). At the same time, others have developed means to

\(^4\)To provide a selection of examples: MPEG-1 [13], MPEG-2 [8], MPEG-4 [9], AVC (Advanced Video Coding) [14], SVC (Scalable Video Coding) [15], SLS (Scalable-to-Lossless Audio Coding) [16], VC.1 (a Microsoft-developed video codec, supported by Blu-ray and other high-definition video formats) [17], Real [18], Flash [7], and Quicktime [5].
create and exchange *metadata*, and this has given rise to new web-based applications
that have experienced phenomenal growth (for example Flickr [19], YouTube [20],
and Gracenote [21]/freedb [22])\(^5\).

A different way to conceptualise multimedia is that metadata is merely an *alternative
abstraction or representation of the media to which it relates*: a different interpreta-
tion, perspective, or view of the same underlying data. In fact, a particular instance of
a piece of content—say one that is encoded using a specific profile of MPEG-4—is
itself clearly just one particular abstraction of the underlying data, since an equiva-
lent (yet distinct) abstraction can be arrived at by using some other type of encoder.

So too, in the uncompressed domain, there are many different ways to represent the
same physical (analog) signal. While Pulse-Code Modulation (PCM) is the predom-
inant way to digitally represent audio, it is by no means the only representation [23],
and video also has several common modes of uncompressed digital representation,
including YCrCh\(^6\) and RGB\(^7\) [24].

Storage devices use yet another abstraction for multimedia data: bytes (or ultimately,
bits). Communication channels, on the other hand, often consider media data in par-
ticular “chunks” (e.g. Packetised Elementary Streams (PES) in MPEG-1/-2 or Net-
work Abstraction Layer (NAL) units in H.264/AVC) so as to limit the consequences
of packet loss [25].
It is proposed that there is in fact a hierarchy (of sorts) between these abstractions (Figure 1.2\(^8\)), in terms of the complexity of use, and also the amount and types of information that may be carried (the \textit{semantic content} of the abstraction). Furthermore, as the level of abstraction increases, so does the dimensionality at that level of the hierarchy: there exist a greater number of alternative abstractions by which the semantics of the content can be accessed (examples of this are discussed below).

This integrative view of multimedia provides a number of useful insights which form

---

\(^5\) Although this metadata is intrinsically tied to the media it describes, it is typically generated independently, by manual input, by the content creation software, or by processing the media in the uncompressed domain (e.g. identifying shot or scene boundaries in video).

\(^6\) \(Y\) – Luminance, \(Cr\) – Red-difference chrominance, and \(Cb\) – Blue-difference chrominance

\(^7\) Red, Green, Blue

\(^8\) this framework has some overlap with the Multimedia Metamodel described in [26], although the latter work addresses only the upper three layers of Figure 1.2, and also device capabilities.

---
Building Babel

the basis for the remainder of this thesis. These insights are detailed below (Section 1.4 on page 9). First, however, Sections 1.2 and 1.3 explore this framework in more detail.

1.2 Many layers of abstraction

At the lowest level, multimedia data is represented by a stream of bits. This level of abstraction (Figure 1.2) provides no semantic content, but allows a digital device to store and retrieve the data. Bit streams are typically divided into groups of 8 (i.e. a byte, or octet) for addressing. This higher layer of abstraction provides a second dimension (in a limited sense) to the one-dimensional bit stream.

Digital data formats (including multimedia codecs) group bits or bytes into fields, which are given particular semantics by the format. A collection of fields is still essentially two-dimensional, but a third dimension is added to this by grouping fields into higher-level syntactic structures such as headers, packets, frames, and so on (Figure 1.2).

As discussed, communications channels typically segment media data into predetermined syntactic structures (often known as packets, which may or may not correspond to the packets defined by the encoding format). At this level of abstraction, a further dimension arises, by virtue of the numerous possible channels and their packetisation mode(s).
Higher-level structures in Figure 1.2 have semantic significance independent from any particular digital representation of the structure. Examples of such semantic structures are a picture or frame from a video, or a group of audio samples. This level of abstraction forms the intersection between the many alternative (un)compressed encoding formats, and also the various options within a single encoding specification. In this sense, encoding, decoding, transcoding, and so on can all be thought of as transformation of the domain of the data (analogous to the more familiar notion of transformation between the (spatio-)temporal and frequency domains). Although its representation may change, the underlying content does not.

Higher levels of the hierarchy are occupied by various types of metadata. Just above the media itself are what are typically referred to as “low-level” or feature metadata: “this picture is predominantly red”; “the object at \((x_0, y_0)\) in this picture moves to \((x_1, y_1)\) in the next picture”; “this sound has an audio signature of \(z\)”. At this level, there is a profusion of added dimensions, because the same content can be simultaneously described in a large number of ways.\(^9\)

Low-level metadata is essentially possible to compute from the underlying data, yet it is typically less relevant to human observers. Much research has gone into bridging the so-called semantic gap from low- to high-level metadata that is more relevant, such as “this is a picture of a rose”; “Josie throws the ball to Philip”; or this audio is

\(^9\)As an aside, it should be noted that established mechanisms for computing many of these dimensions usually operate on data in the uncompressed domain. In other words, in order to extract metadata from content that is compressed, it is necessary to first transform this media data to an uncompressed representation, and then transform that into the desired metadata representation. This operation is typically expensive (in terms of computational complexity), and infeasible in some situations (e.g. the real-time streaming of media data).
“Walter Meinhof playing Beethoven’s Moonlight Sonata on a Grand Piano”. Some notable achievements notwithstanding (see for example [27]), high-level metadata still remains the domain of manual annotation. Despite this, such metadata is still fundamentally just another alternative abstraction, of the same underlying data (the piece of content).

Perhaps highest on the hierarchy are the experiential and subjective characteristics of content, such as “this is a terrible rendition of Moonlight Sonata”, or “the movie is very sad”. While the content of any of these abstractions is completely dependant on the viewer, and may well vary for the same viewer at different times, they are nonetheless relevant abstractions of the same underlying data. Furthermore, they carry the greatest level of semantic content, by moving beyond describing what the content is, to describing the effect that it may have on observers. Any such effect is in a sense the sum-total of the impact of lower-level characteristics (whether or not these characteristics are consciously observed).

1.3 Describing the structure of each abstraction

Every abstraction in this framework has a precise structure that allows an observer (computer or human) to correctly interpret that view. At the bottom of the hierarchy, the structure of both bits (every field has a value of 0 or 1), and bytes (every field has 8 bits) is very simple. Further layers (up to the picture level) successively impose more structure around that of the layer below it.
Languages have been devised to represent the schemata of some of these structures. The Bitstream Syntax Description Language (BSDL) Level 1 [28] allows schemata to be written to describe the organisation of bits and bytes into fields. BSDL-2 [29] schemata describe how these fields are grouped into packets, frames and complete coding formats. BSDL, along with other schema languages such as Flavor [30], provide the means to specify the structure of a complete coding format in a way that is machine-processable. In other words, given its BSDL schema, a processor is able to parse any given bitstream, which is a necessary step for any subsequent processing.

Current metadata specifications typically\(^\text{10}\) use XML, for which the structure can be described using XML Schema [31]. However, people are increasingly looking to ontological schema languages (such as RDFS [32] or OWL [33]) to enable metadata to be more generally useful, beyond the specific purpose for which it was created. In other words, RDFS and OWL promise to make it easier to move between abstractions within the high-level semantic space.

\[1.4\] Implications of the model

The conceptual framework described above highlights numerous opportunities to “Free multimedia... from hard-coded formats”, and several of these form the basis for the remainder of the thesis:

\(^{10}\text{but not exclusively; see Chapter 5 on page 127 for an example of the use of BSDL schemata for binary metadata}\)
1.4.1 Surveying the site

A review of the literature

Chapter 2 on page 19 reviews the relevant literature for various aspects of the model;

1.4.2 Laying the foundations

A universal metasyntax for multimedia

A means of describing the structure of multimedia content and metadata is a central component in enabling machines to interact with content without requiring hardwired knowledge about content formats. A number of approaches to this are identified in the literature (see Section 2.1 on page 20), although all have a number of shortcomings when used to describe modern encoding formats such as H.264/AVC. Chapter 3 on page 63 proposes several extensions to the BSDL language to remedy these shortcomings;

1.4.3 Babel’s formwork

Format-independent streaming and delivery

According to the model, alternative coding formats are simply different abstractions of the same underlying data. Furthermore, the steps involved in mapping content to some delivery channel are conceptually independent of either the coding format or the channel type. However, current delivery paradigms typically require custom-built single-purpose software for every combination of codec and channel that is to
be supported. This $O(n \times m)$ rate \(^{11}\) becomes increasingly difficult to maintain as the diversity of internet-connected devices and content formats continues to grow. In response to this, Chapter 4 on page 95 develops a framework for format-independent multimedia streaming and delivery, that exploits the consistency of the underlying process of content delivery.

1.4.4 Erecting the tower

Applications for format-independent streaming and delivery

Chapter 5 on page 127 extends these concepts and addresses a number of specific application scenarios:

- A format-independent streaming server;
- Generic multimedia syntax translation; and
- Virtual container assembly.

Section 4.1.1 on page 98 introduces these scenarios as part of the motivation for a format-independent delivery framework, while Chapter 5 shows how they may be fulfilled using the Bitstream Binding Language.

\(^{11}\) \(n\) – number of codecs, \(m\) – number of channels
1.4.5 What are you trying to say?

Format-independent semantic-aware streaming and delivery

Numerous abstractions at higher layers in the hierarchy are useful to lower-level processes. For example, the delivery of multimedia data can be improved by knowing which parts of the content are more meaningful to a consumer and which are less so. The delivery process could then devote more resources to the parts of the content that carry greater meaning. Chapter 6 on page 153 describes a generic mechanism to allow delivery nodes to access such higher level metadata, along with other examples of use.

1.4.6 Scattered to the winds

Conclusions and Future work

Chapter 7 on page 201 offers some concluding remarks, and discusses some of the directions in which this work may be extended.

1.4.7 Tearing it all down again

Toward format-independent decoding

Finally, Appendix A presents part of a framework that simplifies the process of moving from a point in the hierarchy representing compressed media content, to a point representing uncompressed content (i.e. decoding). This framework is reconfigurable: the syntax of the content as well as the structure of the decoder can be varied.
to suit the requirements of the content or channel. This work represents an important part of format-independent media delivery, and Appendix A describes both the specific contributions proposed by this thesis, and the larger project of which they are a part.

1.5 Summary of contributions in this work

This section lists the primary contributions that this thesis makes to the body of knowledge regarding multimedia. They are sorted in order of appearance, with the corresponding chapter references and associated publications shown in parenthesis. Citations refer to the bibliography in Section 1.6 on page 15.

- Performed an analysis of the Bitstream Syntax Description Language (BSDL), and identified shortcomings in its use with modern coding formats such as H.264/AVC that lead to $O(n^2)$ complexity with regard to bitstream duration (Chapter 3 on page 63) [Thom 06a] [Thom 07];

- Proposed a means to declare variables within BSDL to allow H.264/AVC streams to be processed in $O(n)$ time (Chapter 3) [Thom 06a] [Devi 07];

- Devised a way to make BSDL extensible so that it is able to describe new coding types, such as Exponential-Golomb codes in H.264/AVC (Chapter 3) [Thom 06a] [Thom 07] [Devi 07];

- Introduced the ability to use BSDL to succinctly describe fields having a dy-
namic length in bits. Such fields are prevalent in H.264/AVC and MPEG-4 (Chapter 3) [Thom 06a] [Devi 07];

- Analysed available techniques for rich media streaming and delivery, and identified requirements for a format-independent approach to this problem (Chapter 4.1.2 on page 101) [Thom 05];

- Created a format-independent framework for multimedia streaming and delivery that addresses the issue of the proliferation of content and metadata formats (Chapter 4 on page 95) [Thom 06d] [Thom 06c] [Thom 07] [Burn 07] [Thom 08];

- Identified a way to unify the addressing (selection, identification) of fragments of media and metadata regardless of whether the underlying format is binary or XML. This simplifies the description of rules for including content in delivery packets (Chapter 4) [Thom 07] [Thom 08];

- Created and validated a means to translate metadata between alternative syntaxes to provide greater interoperability. This allows metadata stored as ID3 (for example) to be translated to another format (e.g. MPEG-7, or iTunes) using a generic (format-independent) processor (Chapter 5 on page 127) [Thom 06b];

- Proposed the concept of Semantic Distortion (SD) that can be used to systematically incorporate knowledge about content semantics into the delivery process (Chapter 6 on page 153) [Thom 07a] [Thom 09];

- Extended Chakareski’s R-D Hinter to encompass SD (Chapter 6) [Thom 07a]
[Thom 09];

- Created an ontology for SD that enables it to be extracted from semantic meta-data and then combined with sample distortion (Chapter 6) [Thom 09];

- Designed a format-independent architecture for the hinter, which operates by extracting all format-specific details into declarative data. It is argued that this is imperative to allow the increasingly diverse range of formats and devices to interoperate (Chapter 6) [Thom 09];

- Created a semantic-independent architecture for the hinter (Chapter 6) [Thom 09];

- Proposed the concept of a self-describing bitstream syntax. This is part of a Reconfigurable Media Coding (RMC) solution that greatly improves the flexibility of multimedia coding to adapt to content, network and terminal conditions (Appendix A) [Thom 07b] [Luca 07] [ISOI 07]; and

- Identified a declarative transformation for a self-describing bitstream syntax schema into an executable module that can parse the content, providing it to other RMC modules (Appendix A) [Thom 07b] [Luca 07] [ISOI 07].

1.6 Publications

The work contained in this thesis has been published in the following articles:
1.6.1 Journal Articles


1.6.2 Conference Papers


1.6.3 MPEG Standards


1.6.4 Patents

Chapter 2

Surveying the site

A review of the literature

This chapter surveys the existing literature with the aim of mapping the terrain on which to build the contributions made in later chapters. Furthermore, this chapter identifies work which could, in some sense, be seen as providing similar functionality to that proposed here, and highlights any relative advantage of one approach or the other.

As discussed, languages to describe the syntax of media content are at the core of format-independent processing. These are reviewed in Section 2.1 on the next page, and form the foundation of the work in the remaining chapters of this thesis. Section 2.2 briefly examines several other languages related to this task.

Before developing mechanisms for format-independent streaming in Chapter 4 on page 95, Sections 2.3.1 and 2.3.2 look at existing approaches. Section 2.3.3 explores multimedia package formats, which provide an alternative format-independent delivery mechanism to streaming, and this mechanism is considered in Chapter 5.
The remainder of this chapter surveys recent coding formats which offer the ability to discard portions of a bitstream while retaining the ability to decode what remains (Section 2.4 on page 44). Approaches have been developed to exploit such bitstreams to better match terminal/network capabilities (Section 2.4.2 on page 47) or optimise content delivery in the face of sub-optimal conditions (Section 2.4.3 on page 53). Chapter 6 combines such approaches with algorithms to adapt content according to its semantics (Section 2.4.4 on page 55) to create a general framework for semantic-aware, format-independent content delivery.

2.1 Syntactic Metalanguages

One of the fundamental requirements for “freeing multimedia... from hard-coded formats” is to have some language that can describe the structure of any given format, in a way that a computer can understand. Such a meta-language allows a system to be able to perform many syntactic operations on a piece of content, without knowing a-priori what form the content will take. Indeed, all of the chapters in this thesis are centred around the use of a syntactic metalanguage to perform some task in a format-independent manner—be it delivery (Chapter 4), (un)packaging (Chapter 5), semantic-aware rate distortion optimisation (Chapter 6),
or decoding (Appendix A). In terms of the hierarchy introduced in chapter one, Syntax-
tactic Metalanguages provide a schema that describes how to assemble bits and bytes
into fields, and fields into higher level structures (Figure 2.1).

Syntax description is a mature field that has its roots largely in programming lan-
guage specification. The (Enhanced) Backus-Naur Form (EBNF) [34] is a notation
that has become the de facto standard for specifying the syntax of programming lan-
guages, although it has many variants. Some of the fundamental operators of EBNF
are alternative (|), sequence (,), and repetition (*). Using these and other operations
one may form production rules that define named nonterminal symbols as sequences
of terminal symbols (alphanumeric constants enclosed in quotation marks) and other
nonterminal symbols [35] (see Figure 2.2a for an example).

Syntax is, in fact, a fundamental aspect of almost any form of communication, and
metasyntax notations have been developed for other domains as well. For example,
Abstract Syntax Notation One (ASN.1) [36] is often used to specify network proto-
ocols, and XML Schema [31] or Document Type Definitions [37] constrain the syntax
of XML documents. ASN.1 has a similar approach to EBNF in that it is based
on the formation of symbolic predicates. However, ASN.1 is more verbose in its
specification—for example—of SEQUENCEs and CHOOSEs (Figure 2.2b). XML
Schema is the basis of BSDL and is considered further at a later point (Chapter 3).

Specification of multimedia syntax, on the other hand, has traditionally used ad-hoc

---

1XML: eXtensible Markup Language [37], a simple yet expressive syntax for annotating (marking-up) text, which is increasingly used as the de-facto standard for communication on the Worldwide Web.

2Note: code snippets in the text, figures and listings of his thesis are formatted like this.
notations (Quicktime, for example [5]), some of which are loosely based on EBNF (such as AVI (Audio-Video Interchange) [38]). More recently, two multimedia-specific syntax metalanguages have been proposed: Flavor and BSDL. Flavor [39] uses C++/Java-like expression to specify bitstream syntax for automatic parser generation and is described in Section 2.1.2 on page 25. The Bitstream Syntax Description Language (BSDL) [40] (described further in Section 2.1.1), on the other hand, is an extension of XML Schema. Both languages specify how atomic data-types map to binary symbols, in a fashion analogous to the languages cited earlier. However, Flavor and BSDL add control-flow constructs to specify how particular instances of a given format are to be parsed. EBNF, ASN.1 and the like do not provide this.

To highlight the differences between syntax description languages, Figure 2.2 on the facing page shows descriptions of the same part of an MPEG-4 Visual bitstream [9]. ASN.1 explicitly separates content (abstract syntax, shown in the figure) from encoding (not shown). This is in recognition of the fact that identical message content may be encoded differently depending on context (for example, a more efficient binary encoding may be preferred for low bandwidth applications). This distinction is indeed valid for multimedia: the same content could be encoded in H.264, or in MPEG-4 Simple Profile, for example. However, the distinction is not relevant at the level of individual symbols at which it is made in ASN.1. For multimedia, abstraction of content from encoding at the symbol level significantly adds to complexity, without improving portability.

In order to parse raw content of a particular format, additional information is nec-
VideoObjectLayerType ::= [ longHeader | shortHeader ],
[ VideoObjectPlaneType ]
longHeader = VOLStartCode, {"...and so on..."}
VOLStartCode = StartCodeType
StartCodeType = 4 * hex-digit
hex-digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' |'A'|'B'|'C'|'D'|'E'|'F'

(a) EBNF

class VideoObjectLayer() {
int[32]* next_bits; //look-ahead only
if(next_bits == 0x00000120) {
short_video_header = 0;
// ... = 1;
// ... 
}
{
VideoObjectPlane();
( 32 )* next_bits; 
} 
(next_bits ==  0x000001B6);  
}

(b) ASN.1

<complexType name="VideoObjectLayerType">
<sequence>
<choice>
<group ref="longHeader" bs2:ifNext="00000120" ...
<sequence>
<element 
name = "VOLStartCode" type = "SCType" />
<!-- ...and so on... -->
</sequence>
</group>
</choice>
</sequence>
</complexType>

(c) Flavor

<complexType name="VideoObjectLayerType">
<sequence>
<choice>
<group ref="longHeader" bs2:ifNext="00000120" ...
<sequence>
<element 
name = "VOLStartCode" type = "SCType" />
<!-- ...and so on... -->
</sequence>
</group>
</choice>
</sequence>
</complexType>

(d) BSDL

VideoObjectLayerType := SEQUENCE [ header VOLHeaderType, vops SEQUENCE OF vop]
VOLHeaderType := CHOICE {
longHeader LongHeaderType,
shortHeader ShortHeaderType}
LongHeaderType := SEQUENCE {
volStartCode StartCodeType, --...and so on--
}
StartCodeType := OctetStringType [SIZE(4)]

Figure 2.2: These examples show a small part of the MPEG-4 Video syntax in several of the numerous syntax description metalanguages.

Figure 2.2: These examples show a small part of the MPEG-4 Video syntax in several of the numerous syntax description metalanguages.

Figure 2.2: These examples show a small part of the MPEG-4 Video syntax in several of the numerous syntax description metalanguages.

Figure 2.2: These examples show a small part of the MPEG-4 Video syntax in several of the numerous syntax description metalanguages.

...essary beyond a description of its syntax. This may be seen, for example, in EBNF (Figure 2.2a) and ASN.1 (Figure 2.2b), which specify that a Video Object Layer (VOL) object may contain either a long header or a short header, but not how to tell which is actually present within a given bitstream. In contrast, Flavor provides this information via an if block, and BSDL using an bs2:ifNext attribute (or others, see Section 2.1.1).

2.1.1 Bitstream Syntax Description Language

The BSDL Schema language [29] extends XML Schema [31] with mechanisms to specify the bit-level equivalent for XML data types, as well as conditional structures to describe parsing of a bitstream. As a result, a BSDL Schema elegantly describes
both the bitstream, its XML representation (the Bitstream Syntax Description, or BSD), and the translation between the two. Furthermore, BSDL has explicit byte-range data types (with mechanisms to identify the end of a range). This enables a BSDL Schema designer to expose only those binary fields explicitly necessary for their purpose, and aggregate the rest of the content type into opaque byte-ranges. Given the large number of repetitive data fields in any multimedia bitstream, this feature can greatly improve efficiency.

BSDL was originally designed to enable adaptation of scalable multimedia content in a format-independent manner, that is, using adaptation software that did not possess detailed knowledge of the content format it was adapting (Figure 2.16 on page 52). This was achieved via an XML view of the content, hence the choice of XML Schema.

With regard to Figure 2.2d, the structural features of an XML Schema (known as particles): `<choice>` (between one or more sub-particles), `<sequence>` (of sub-particles), `<element>` and `<group>` (which creates a named global particle). Elements are given a type, which is defined by either a `<complexType>` (the element contains another particle), or a `<simpleType>` (the element contains textual or binary content). The facet `<maxExclusive>` defines the number of bits read by integer types, and `<length>` the number of bytes in a string or hexBinary type.

In BSDL, simple types convey binary data, while complex types create structure around simple types. `bs2:ifNext` is one type of parse-control, specifying that the attached particle is parsed only if the next bytes to be read from the bitstream cor-
respond to the ones in the attribute value. This is commonly used for specifying start-codes; the start code for an MPEG-4 VOL object is 0x00000120.

### 2.1.2 Flavor/XFlavor

Flavor [41] (Formal Language for Audio-Visual Object Representation) was developed to enable automatic generation of bitstream parsers. Flavor uses a C++ like syntax to specify the structure of a bitstream on a field-by-field basis, providing declarative if-else, switch, for and while statements to describe complex bitstreams.

In the example (Figure 2.2c on page 23) an if statement is used to decide upon header type (the * indicates a look-ahead type), and a subsequent do—while statement used to parse VideoObjectPlane (VOP) objects for as long as the next bits indicate a VOP start code. In Flavor, parsable variables (ones that read their value from the bitstream) are types with a bit-length specification (VOLStartCode for example). Non-parsable variables can be used to perform internal calculations without directly reading from the bitstream.

A recent extension (XFlavor [30]) is able to produce an XML description of the syntax out of the process, which could potentially be utilised as metasyntax. However, (X)Flavor outputs an XML element for every field of a bitstream, resulting in extremely verbose BSDs, making description of any real-life multimedia bitstream too verbose to be feasible [42].
2.1.3 BFlavor

In response to the verbosity of XFlavor, De Neve et al propose an extension to XFlavor [42] that attempts to harmonise it with BSDL—particularly with regard to the latter’s byte-range functionality. This extension—BFlavor—is targeted specifically at H.264/AVC, however it may be used only for a small subset of possible H.264 bitstreams: “the [B]Flavor description only allows to process H.264/AVC bitstreams that contain one Sequence Parameter Set (SPS) and one Picture Parameter Set (PPS)” [42]. This precludes all SVC bitstreams, in which each layer has its own parameter sets, along with many AVC streams. As a result, it is unsuitable as a truly generic (format-independent) syntax metalanguage.

2.1.4 Comparison between BSDL and XFlavor

BSDL and XFlavor both provide a solution for describing the syntax of binary media content. However, in the context of the particular work in this thesis, BSDL has a number of advantages which make it more desirable:

- a BSDL Schema contains all the data required to both parse a bitstream and operate on the XML, whereas XFlavor requires a Flavor description for the former and a separate XML Schema for the latter. BSDL, therefore, represents a more compact, elegant solution;

- because it extends XML Schema, it is easy to use BSDL to directly identify a fragment of binary content using languages designed to identify XML frag-
Surveying the site

ments (the XML Path Language (XPath) [43], for example). Identification of binary content fragments is fundamental to the work in Chapters 4 and 6.

- BSDL is defined by an international standard [40];

- the XML-based syntax of BSDL itself integrates easily with the XML syntax used to describe delivery instructions (see Chapter 4), Semantic Distortion (Chapter 6), and an RMC decoder (Appendix A); and

- there are many tools and languages available for parsing [44–46], transforming [47,48], querying [49] and otherwise processing XML and XML Schema (and hence BSDL). [47]).

Finally, on a purely practical note, the implementation of both systems are highly suboptimal. XFlavor uses a brute-force approach to “automatic code generation” [30] which results in—among other things—deeply nested switch-case groups. For example, the Flavor description of a 256 element Variable Length Code (VLC) table generates XFlavor source code of 288kb with switch-case blocks nested 16 levels deep! BSDL, on the other hand, uses the Apache Xalan XPathAPI [50] to execute XPath queries. This API is very slow and does not scale well to large numbers of queries, as is necessary for the work in this thesis. The latter issue, however, is far more easily addressed than the architectural issues associated with Flavor code generation.
2.2 Media and Metadata Fragmentation

The authors of BSDL as well as other work described in Section 2.4.2 propose the use of the eXtensible Stylesheet Language Transformations (XSLT) language. The XML Path language is a central component of XSLT, and also of BSDL. Consequently, both are discussed briefly below. These languages essentially provide Feature-level metadata (Figure 2.3) that identifies the byte location of fragments of media or metadata.

2.2.1 Extensible Stylesheet Language Transformations

eXtensible Stylesheet Language Transformations (XSLT) [47] is an XML transformation language originally designed for use by XSL, which is a stylesheet language for XML. It has, however, proven to be widely useful in many XML transformation scenarios. For example, the authors of BSDL [28] intend for XSLT to be used on the output XML description, to adapt the bitstreams by removing scalability layers. In other domains XSLT is used to transform XML between differing conceptual models for knowledge representation [51], and to customise distance-education presentations for individual students [52].
XSLT, along with other languages such as XML Query (XQuery) \cite{49}, are proposed in Chapter 6 as possible mechanisms for specifying mapping rules that link content parameters to delivery metadata. See Section 6.3.2.1 on page 169 for a critical analysis of different approaches.

### 2.2.2 XML Path Language

The XML Path Language (XPath) \cite{53} is a language developed by the W3C \cite{54} as a method of addressing parts of an XML document. While designed to meet the requirements of XSLT and XML Pointer language \cite{55}, it has proved to be broadly applicable and is used by many applications needing to identify nodes in an XML document. It is based on a simple path syntax similar to that used by URLs, however XPath also specifies a large set of extensions to this syntax that provide a flexible mechanism of querying XML documents. These extensions include axes (such as ancestor and following−sibling) which declare a direction of search from the context (current) node, wildcards (\(*, /\)), and functions for evaluating strings, numeric expressions, and XML constructs.

In addition to being widely used in XSLT, XPath is also fundamental to the operation of BSDL, and is used throughout the chapters of this thesis.
2.3 Format-independent Media Delivery

Perhaps the most essential application scenario implied by the title of this work is that of a format-independent streaming server. This application (Figure 2.4) uses metadata to describe how to deliver binary content (and other metadata) so that the software and hardware used need not possess detailed knowledge about the content format(s) being delivered. This is motivated by the observation that multimedia technology is developing at an ever increasing rate. New audio, video, and hybrid encoding formats are regularly developed, and the number of devices accessing or processing multimedia content has grown exponentially, as has their variability in terms of available processing power [10]. This diversity hampers interoperability because tools that handle multimedia data are generally required to have custom software written to handle each format. As new content formats are defined, or even as different formats become widely popular, they cannot become useful until software has been written and deployed for the set of platforms which process or consume them, including streaming servers, multimedia gateways, and consuming devices from PCs to mobile devices.

With regard to streaming servers, the following section provides a comparison of various approaches that have been devised in response to this issue, along with a
Figure 2.5: Monolithic Streaming Server

discussion of the gaps that remain. The work in Chapters 4 and 5 subsequently address these gaps by moving format-specific details out of software code and into a schema (i.e. a data file).

More generally, this work also proposes a mechanism for packaging and streaming metadata along with the content it describes, so Section 2.3.2 reviews tools that exist for this purpose.

2.3.1 Streaming Server Architectures

Figures 2.5, 2.6, and 2.7 show a number of possible architectures for a multi-format streaming server. The simplest case (Figure 2.5) has software modules for each supported format to process content of that form and ready it for streaming. When a new content format is developed, additional software modules must be developed and integrated into the streaming server in order to support the new format.
2.3.1.1 Hint Tracks

Quicktime files [5] and the ISO file format [56] (see below—Section 2.3.3.1) provide a mechanism known as “hint tracks” which suggest how a server could stream the content in the file. This means that the streaming server itself (Figure 2.6) no longer needs to explicitly provide software to support each individual content format. Instead, the server may stream any content simply by processing its hint track(s). This architecture significantly increases scalability, since hint track processing is essentially a sequence of byte-copy operations—requiring much less computation than parsing the bitstream to determine how it is to be streamed.

This latter computation is still required, but it may now be conducted offline—and often on a different machine—in a separate hinter application. Consequently, the hinter still requires specific software to process each individual format, and must be updated in order to support new encodings as they are developed. In practice, there are significantly more hinter applications than there are streaming servers. As a result, interoperability for new content formats is yet more difficult, since the number of applications for which new software must be developed is substantially larger.

2.3.1.2 gBSD-based ‘generic streaming’

Ransburg et al have considered this issue, and devised a “gBSD-based generic streaming server” [57]. gBSD—generic Bitstream Syntax Description—is a tool related to BSDL (see Section 2.1.1 on page 23) which uses a single schema to describe all bitstreams. Ransburg et al propose “to use an extended version of the gBSD as a
hint file.” Specifically, the gBSD is extended with a marker to identify Access Units\(^3\) (AU)s and specify a timestamp for each AU.

While the gBSD schema is generic (format-independent), the generation process is not. Generating a gBSD for a piece of content requires specific software which is able to parse the format in question, for which the authors suggest modifying media encoders to output this extended gBSD as a side-band. The “gBSD-based generic streaming server” therefore has the architecture of Figure 2.7 on the following page: streaming itself is generic, but the hinting/encoding application is not—it requires additional software to support new content formats.

Additionally, the identification of access units does not generally provide sufficient information to stream content. Many content formats place additional restrictions on

\(^3\)Access Unit: the smallest unit of data to which timing may be attached.
packetisation below the level of an access unit. For example, the specification for transmitting H.264/AVC over the Real-time Transport Protocol (RTP) [58] places constraints on the fragmenting of NAL\(^4\) units (part of an AU).

Content formats also often require custom header information to be transmitted as part of the stream—for example, H.264/AVC or MPEG-4 over RTP [58, 59]. The fields in the custom header are generally based on the payload, but not included within it. For these reasons, the extended gBSD hint file provided by Ransburg et al does not provide enough information to stream the content.

\(^4\)NAL: Network Abstraction Layer. This is the network-level building block of an H.264 stream
2.3.1.3 Discussion

Hinted Streaming servers and gBSD-based streaming both enable format-independent streaming servers, by offloading the format-specific processing onto an offline process called a hinter. In the former case, the primary motivation for this is increased scalability, however in the latter case it is unlikely this would be achieved since the “hint” metadata is reported at between at between 15 and 550 percent of the size of the content itself [60]. In any case, problems of interoperability are still manifest because formats are still hard-coded in the software or hardware of the hinter. Chapters 4 and 5 propose a solution for both a streaming server and a hinter that can exploit the advantages of hinting, whilst being fully format-independent.

2.3.2 Metadata Streaming

There have been several mechanisms for implementing XML based metadata streaming (Figure 2.8) proposed in the literature. Some, such as XStream [61], relate to the fragmentation of XML for incremental delivery. Others combine this with a binary encoding—e.g. BiM [62] and Millau [63]. Annodex [64] also addresses metadata streaming. However, none of these approaches provide a general framework to attach metadata to media content in a streaming format.
These approaches all provide specific streaming formats. They may potentially be
the output of a format-independent delivery process, that is, this process operates at
a more abstract level—it is intended to allow metadata to be mapped into any desired
format, as required by the specific application.

2.3.2.1 XStream

XStream [61] is a mechanism developed for fragmentation and reassembly of XML
documents in wireless environments. It emphasises the need for decomposition based
on the tree structure of a document, in order to allow the processing of each fragment
before the entire document is received. The controllable parameter of the XStream
approach is a Maximum Transfer Unit (MTU), which is used to assemble fragments
node by node, until the MTU is reached. The context path of each fragment is speci-
fied using a numerical ID indicating the sibling number of each successive ancestor.
For example, an ID of “0/1/2” indicates that the root node of this fragment is the 2nd
child of the 1st child of the root (0) of the document.

2.3.2.2 BiM

BiM [62] (Binary format for Metadata) is a method of binary-encoding XML docu-
ments. In addition, it provides mechanisms for fragmenting XML trees for transport,
and defines Action Units which can add, replace, delete or reset fragments within the
destination tree. Action Units contain a numerical code specifying which action to
perform, and a Context Path locating each fragment within the document.
Please see print copy for image

Surveying the site

2.3.2.4 Annodex

The Annodex [66] format has been developed with the aim of providing hyperlink and search functionality to multimedia data. An Annodex stream consists of a media resource interleaved with XML markup containing annotation and index information (an-nodex). A small XML grammar is specified for this that provides very simple structured and unstructured metadata elements, which are in turn loosely based on a subset of HTML. This grammar is known as the Continuous Media Markup Language (CMML), which marks up a single resource annotation track(s).

Before being interleaved with the content, the CMML specifies the single resource to be presented, <head> information relevant to the entire resource, and <clip>
elements providing metadata that is relevant until the next clip is encountered. Clips may contain hyperlinks, ‘thumbnail’ images, metadata structured according to a specific profile, and unstructured descriptive text. Annodex streams are transported using Ogg [64], with each CMML section encapsulated in an Ogg page. Figure 2.9 depicts a sample Annodex file.

The concepts proposed by Annodex represent a significant enhancement to static video presentations, but its scope is deliberately limited so as to “create a very simple set of tools and formats for enabling time-continuous Web resources with the same powerful means of exploration as text content on the Web” [64].

2.3.2.5 Discussion

A number of these related works discuss decomposition of XML into structural fragments, with varying methods of specifying the original context of each fragment. BiM, in particular, provides a dynamic framework whereby fragments may be up-
dated and deleted, enabling a document to change over the course of its lifetime.

Annodex allows simple metadata to be attached to temporal intervals of a single resource, annotating resource fragments, and allowing the consumer to hyperlink into and out of the content at defined points.

However, all of these operate at a lower level than a general-purpose media and metadata delivery framework. Each could be a particular output format into which the framework places content, but they do not provide the generality to specify how that process should proceed. Such a framework is proposed in Chapter 4.

### 2.3.3 Content and Metadata packages

The preceding sections considered streaming formats and tools for both media and metadata. These works are important both in enabling parts of a format-independent delivery framework, as well as being specific formats that could be output by such a framework. In contrast, this section explores a number of storage formats (Figure 2.10) that together contain a considerable proportion of the content on the internet. This forms another important delivery mechanism, which must be considered in a format-independent delivery framework.

![Figure 2.10: Content/Metadata packages](image-url)
ISO file format

The ISO Base Media File Format (formally MPEG-4 Part 12 [56]) began as the Quicktime [5] file format. Since it was contributed to MPEG and became part of MPEG-4, it has been very widely adopted, both by other ISO standards: AVC, SVC, MAF (see below), and JPEG-2000 [68]; and in the broader community: iTunes, Flash, and the 3rd Generation Partnership Project (3GPP) [69], for example.

ISO files have an object-based structure, via a set of nested “boxes” (Figure 2.11). Each box has a particular semantic, and contains fields to specify box type and size, along with other data fields and/or other boxes specific to the box type. Every ISO file has a file type box (ftyp), which specifies the type of content within the file, and usually movie (moov) and media data (mdat) boxes. The movie box contains many nested boxes which store the low-level metadata necessary to understand the raw content (which is usually stored as raw streams within the mdat box).

Movie metadata is organised in tracks (trak), each one representing an audio, video, hint or other type of elementary stream. Track boxes contain numerous further boxes
describing the track, including sample tables (stbl), timing data, and track-level metadata (meta). Conversely, a moov-level metadata box—if present—contains global metadata such as MPEG-7 descriptions of the content or links to related data.

When an ISO file contains hint data, the movie box will have one or more hint tracks (one for each hinted media stream). Each hint track will contain a sample table, which references hint samples in the same manner that media sample tables reference media samples. There are several types of hint samples, including one that references a static portion of data within the file, and another that indirects via a media sample table to represent some or all of a media sample [56].

### 2.3.3.2 Music player MAF

The Music Player MAF format [70] combines audio, images and MPEG-7\(^5\) [71] metadata within an ISO file to provide an augmented digital music library. Figure 2.12 on the next page shows one of the possible structures for a MAF file. The MPEG-21\(^6\) Digital Item Declaration Language (DIDL) [74] XML document in the meta/xml box describes the relationships between the media content included in the file, which are identified by URI. The Item Information/Item Location (iinf/iloc) boxes provide a link between these URIs and the data bytes representing the media content and metadata, which are stored in the media data (mdat) box. The MPEG-7

---

\(^5\)MPEG-7: the Multimedia Content Description Interface, a very large specification of a wide range of multimedia-related metadata

\(^6\)MPEG-21 is officially termed “The Multimedia Framework” and seeks to provide the glue needed to deliver end-to-end multimedia solutions [72]. The work in Chapter 4 forms the basis of MPEG-21 Part 18 [73].
metadata is also (unusually\(^7\)) stored within the `mdat` box, in a “hidden” `moov` box designed to act as an extra layer of indirection.

A final note regarding the Music Player MAF format is that the audio data is encoded as MP3onMP4 Access Units [70]. This data must be deformatted in order to be rendered by existing MP3 devices, but has the advantage of being directly streamable.

\(^7\)normally there are no boxes within the `mdat` box, only raw data
2.3.3.3 ID3

While the ISO and MAF file formats (above) are to greater or lesser degree generic multimedia containers, ID3\(^8\) [76] is much more constrained, being solely a metadata container format used with MP3 audio files. It is, nonetheless, extremely widely used, and is in fact the de facto standard for embedding metadata in the ubiquitous MP3 audio format. As a result ID3 is important to be considered with other container types.

Syntactically, ID3 is in fact very similar to the ISO format, being made up of nested boxes (or Frames, in ID3 nomenclature) that have a type tag along with a size field. It is in the types of Frame available in ID3 that it differentiates from ISO: with ID3 being concerned with frames such as Title, Album, Track #, Artist, Composer, or Attached Picture (typically Album Art).

2.3.4 Summary

With regard to format-independent media delivery, there exist solutions to parts of the problem, for instance hint tracks that allow streaming servers to push format-specific details onto offline processes known as hinters. A broad conception of media delivery that includes metadata streaming recognises that there exist numerous formats for this task (such as XStream, BiM, or Annodex) but no existing mechanism for assembling arbitrary metadata into these formats without resorting to format-specific hardware or software. Similarly, the ISO file format and its many derivatives, along with

\(^8\text{meaning, simply, “IDentify an mp3” [75]}\)
the ID3 metadata container, are widely used media delivery mechanisms, yet there is no mechanism to (dis)assemble such containers in a format-independent manner.

2.4 Scalable Content and Semantics

* Scalability* refers to a method of encoding the fields of multimedia content into syntactic and semantic structures (Figure 2.13) in such a way that the bit-rate used to transmit it may be scaled downward in one or more steps, without re-encoding. This allows a trade-off to be made between bandwidth and quality\(^9\) (or one of several other dimensions, including spatial, temporal, or chromatic resolution). Scalable content greatly enhances the flexibility of multimedia delivery systems, by (among many other things):

- allowing a server to store a single version of a piece of content, yet deliver it to multiple heterogeneous clients. Each client can receive the content at the best resolution and frame-rate for their particular device [77], considered further in Section 2.4.2;

- improving the ability of a delivery chain to adapt to dynamically changing

\(^9\)typically referred to as signal-to-noise ratio (SNR) scalability
conditions by choosing the best available trade-off between bit-rate and quality
at each point in time [78], as discussed in Section 2.4.3; and

- making it possible to seamlessly transfer a streaming session that is in progress
  from one terminal to another dissimilar terminal without interruption [79]. For
  example, a user may be watching content on a mobile terminal and wish to
  transfer the content to their TV set when they arrive home. It is left to [79] to
  explore this application further.

Mechanisms that have been developed to use semantics to improve delivery optimi-
sation are also considered (Sections 2.4.4 and 2.4.5). First, however, Section 2.4.1
describes some of the more prominent approaches to scalable formats themselves,
found in the literature.

2.4.1 Scalable Content Formats

2.4.1.1 Scalable Video

Early work in video scaling focused on providing feedback of network conditions
to the encoder, so that it may adjust various parameters to alter the output bit-rate
[80]. This approach targets real-time applications such as video-conferencing where
encoding and delivery operate concurrently, but is unsuitable for delivery of pre-
encoded content. Other work of a similar vintage proposes so-called “dynamic rate
shaping” by selective truncation of DCT coefficients from encoded pictures [81].
While this allows adaptive delivery of stored media, it was found to yield poor results
More recently, MPEG published extensions to the MPEG-4 video format, known as Fine Granular Scalability (FGS) [82]. FGS aims to provide SNR and temporal scalability without significantly increasing the computational complexity of either decoding or streaming. This is achieved by encoding video content into a base layer and a single enhancement layer. The latter performs bitplane DCT coding on the base layer residual to achieve fine-grained scalability.

The H.264 video codec [12] introduced a new structure for motion compensation reference picture organisation, providing significantly more control over frame interdependency and thus temporal scalability. More recently, amendments have been introduced to this standard that provide fine-grained SNR as well as spatial scalability [83] (often referred to as SVC - Scalable Video Coding). In addition to providing fine-grained SNR scalability via a similar means to FGS, SVC performs coarse-grained SNR and spatial scaling using separate coding layers and inter-layer prediction techniques. Also, to improve the poor coding efficiency typically observed in FGS, motion compensation prediction for the enhancement layer is performed using complete reference pictures, rather than only their base layer.

### 2.4.1.2 Scalable Speech and Audio

Audio and speech generally have much lower bit rates than video. However, for low bit rate mobile terminals, the audio or speech signal becomes a significant component in the perception of audio-visual content [84], and thus the ability to gracefully scale
its bitrate is important. Earlier approaches to scalable audio coding focused on multi-rate audio coders, such as MPEG-4 AAC [85], while more recent scalable coders aim to provide for fine-grained scalability, such as the recently standardised MPEG Scalable to Lossless (SLS) coder [86].

Approaches to scalable speech coding include the multirate coders, such as the recent adaptive and variable multirate wideband speech coders of [87,88], with recent work focusing on fine granular scalable coders, such as that described in [89]. Many of these scalable coders are designed to maximise the perceptual quality at each bit rate. More recent research has focused on scalable coders that are resilient to packet losses in networks, such as the multiple description based coders described in [90,91].

### 2.4.2 Content Adaptation

Dynamic adaptation of scalable multimedia content has received considerable attention in recent literature [29,92,93]. This uses metadata about the delivery scenario to adapt the way that the content is delivered (Figure 2.14). It simplifies the task of streaming multimedia to a wide range of devices with varying processing power and memory capacities, across networks with divergent bandwidth and error characteristics. In particular, Devillers [29] proposes a generic adaptation
architecture which allows processing nodes to perform the adaptation without specific knowledge about the bitstream being adapted. Such an architecture facilitates interoperability for new content formats by enabling adaptation without modifying existing infrastructure. This feature is crucial, since the number of possible formats for multimedia content is large and growing. To be widely useful, an adaptation framework must be able to cope with content irrespective of the format in which it is encoded.

This generic adaptation process is illustrated in Figure 2.15 on page 51. It is based on performing the adaptation on a high-level representation of the bitstream rather than on the bitstream itself. This high-level representation is known as a Bitstream Syntax Description (BSD), and is extracted from the bitstream via a generic processor, aided by a description of the content format (the Bitstream Syntax (BS) instructions) which a BSD to Binary processor applies to the bitstream in order to execute the adaptation.

Hutter et al. [94] build on Devillers’ architecture to support both distributed and dynamic adaptation. The former denotes that the adaptation operation can occur at any (and possibly multiple) points in the delivery network. Dynamic adaptation, on the other hand, is where the adaptation process can react to dynamically changing parameters in real time. In [94], the scope of such parameters is limited to network bandwidth estimates.

Figure 2.15 on page 51 depicts the architecture of this framework. Fragments of metadata are delivered alongside content fragments, to describe the high-level structure of the multimedia. This allows an XSLT transformation to selectively prune parts
of the structure to effect the adaptation, as well as adjust any header fields affected by the pruning. Feasible adaptation operations are, for example, downsampling by removing non-reference frames, or removal of scalability layers.

The XSLT transformation can be parameterised to provide multiple adaptation operations, in which case additional metadata is delivered with the content to detail the available options, their effects, and constraints. Such options may be static, or may be updated over the course of the delivery. An optimisation engine is used to compute the optimum adaptation(s) to perform upon the bitstream to meet dynamic bandwidth constraints, as well as static constraints on the network, terminal device and/or user. Static constraints are provided to the adaptation node via other metadata.

The transformed metadata is subsequently fed to an XML to Binary module, which enacts the transformation on the content itself. Performing the transformation in the XML domain in this way allows all of the modules within the adaptation node to be generic (format-independant).

There are a number of issues with the architecture as it stands. Firstly, the syntax metadata is reported at between 15 and 550 percent overhead\(^\text{10}\) [60], although De Schrijver et al suggest that compression may be used to significantly reduce this amount. Compressing the metadata in this way effectively trades space for time, adding complexity to the adaptation process. An alternative approach [95] recognises that the high-level structure of content fragments is in fact very regular, and conforms to a schema that is part of the normative description of the content format.

\(^{10}\text{as a proportion of the size of the content}\)
Rather than transmitting the structural metadata and incurring the large overhead, the
schema for the format as a whole may be delivered, and the structural metadata gen-
erated on-the fly in the adaptation node (Figure 2.16 on page 52). Like compression,
this approach trades time for space, but reduces the space requirement to a negligible
amount. Secondly, the architecture uses a wide range of metadata about the usage and
delivery environments, but its examination of the content itself is limited to syntac-
tic characteristics: scalability layers and bit-rate [94]. Clearly, these parameters are
fundamental to the adaptation process. However, as discussed in Chapter 6, semantic
knowledge of the content can lead to greatly improved adaptation performance.

Leopold et al. [96] present an alternative view of the MPEG-21 adaptation frame-
work. Instead of utilising format-independent processors, they envision a web ser-
vices approach to adaptation, where nodes communicate the adaptation operations
that they are able to perform (including all necessary details about supported formats
and constraints). For example, one node could support an operation that reduces the
resolution of JPEG, GIF and TIFF\textsuperscript{11} images, while another node can convert from
color to greyscale. A path-finding algorithm is then used across the state-space of
all of the atomic operations, to find the sequence of steps necessary to complete the
required adaptation. In [100] this concept is extended to use Semantic Web Ser-
vices, using semantic markup languages (such as RDFS and OWL) to describe the
supported operations. This work makes use of semantic tools to describe adaptation
operations, but not to describe the content itself.

\textsuperscript{11}JPEG—Joint Photographic Experts Group. JPEG is the defacto standard for storage of consumer-
grade photographs [97]; GIF—Graphics Interchange Format [98], a popular format for web-page
graphics; TIFF—Tagged Image File Format, another image format [99].
Figure 2.15: Dynamic Adaptation using MPEG-21 (adapted from [94])
Figure 2.16: Bandwidth-efficient Dynamic Adaptation using MPEG-21 (adapted from [94])
2.4.3 Rate-Distortion Optimised Packet Scheduling

In the preceding section, Content Adaptation seeks to scale content primarily on the basis of external characteristics such as terminal and network capabilities. In contrast, Chou et al. [101] approach the optimisation of delivery of scalable content by looking at the content itself in a very fine grained manner (Figure 2.17): specifically, how much distortion would each individual Data Unit produce if it was not delivered? That is, the process of deciding if, and when, to send each atomic unit of scalably coded media is a problem in rate-distortion optimisation.

Put differently, given the constraint of a dynamically changing channel bandwidth, which parts of the media should be delivered to minimise the total distortion of the content at the receiver? Using classical optimisation techniques, optimal distortion-rate performance $D(R)$ may be achieved by minimising the Lagrangian $D + \lambda R$ for some $\lambda$ [101]. This is in turn based on error probability-cost functions for each unit of data, where errors are characterised in such a way as to encompass bit error rate, packet loss, and delay (such that the packet is too late to be useful).

Furthermore, the formulation of distortion must consider the interdependencies between Data Units, since descendent packets (e.g. enhancement layers in FGS or
SVC, or any motion-compensated frame) generally cannot be decoded if their ancestors are not received. This dependency is modelled as a directed acyclic graph [101] and is used to compute the distortion $D$ of a set of packets $\pi$ with error probabilities $\epsilon$:\footnote{where $D_0$ is the distortion when zero packets are received. $(l' \prec l)$ indicates that $l'$ ranges over the set of ancestors of $l$.}

$$D(\pi) = D_0 - \sum_l \Delta D_l \prod_{l' \prec l} (1 - \epsilon(\pi_{l'})) \quad (2.1)$$

Using this model, the optimal set of transmission decisions is completely characterised by

- the dependence graph;
- the set of distortion increments $\Delta D_l$;
- the set of packet sizes; and
- the error-cost functions.

All but the final parameter are features of the media, and error-cost is a characteristic of the channel. The dependence relations $l'$ complicate the subsequent optimisation process, but it may be solved using an iterative descent algorithm [101].

Although the algorithm proposed by Chou is theoretically optimal and suitable for certain applications, Chakareski et al. [102] note that this optimality comes at the cost of significant computational complexity. Indeed, the optimal packet schedule must
be recomputed with every packet (and possibly more often than that). Consequently, Chakareski proposes a low-complexity approximation of the lagrangian optimisation problem by considering the distortion imposed by packet loss on subsequent packet distortions to be additive, instead of the acyclic graph used by Chou. The cost of this approximation is reported as between 0.5 dB and 6 dB, with better performance observed at the higher end of the bitrate range.

Eichhorn [103] suggests an alternative approximation technique that essentially takes the opposite approach: whereas Chakareski ignores actual dependencies, Eichhorn ignores actual distortions, and suggests that dependency alone may be sufficient.

Finally, Cranley and Murphy [104] use measures of Spatial Information (SI) and Temporal Information (TI) along with subjective measurements to derive an Optimum Adaptation Trajectory for spatio-temporally scalable media by making consecutive trade-offs between the two dimensions.

### 2.4.4 Adaptive Delivery using Semantics

Several approaches have been proposed to adapt or deliver content according to certain semantics (Figure 2.18). Bertini et al. [105], Xu et al. [106], and Baba et al. [107] all argue that applying the same adaptation operation to different parts of a multimedia presentation will have differing effects in the perceptual quality of the presentation as a whole. More specifically, Bertini and Xu both propose adaptation on the basis of semantic classification of sporting events into categories such as Shot on Goal, Corner (for soccer) [105], or Shot, Foul, Penalty (for Basketball) [106], among
Surveying the site

others. User preferences are used to prioritise the categories, and this priority information is then used to guide the adaptation. In other words, given a bandwidth constraint such that the full content can not be delivered, the adaptation engine will reduce the bit rate of lower priority sections before those with higher priority.

This semantic metadata can be considered as very high-level, and coarse-grained. In other words, it identifies relatively large segments of content, using concepts with a high level of abstraction from the digital representation of the content. In the first case, heuristic methods are proposed to automatically classify content segments, with a precision of 83% to 96% [105].

Baba et al. [107] propose adaptation of speech signals on the basis of a much lower-level semantic concept: sound level. They argue that regions of (relative) silence within a speech signal carry little or no semantic information, and as such may be truncated during playback. In fact, this feature of speech\textsuperscript{13} may be used to guide adaptation, allowing regions of silence to be constrained to a zero bit-rate (or as close as the scalable codec will allow) with no perceptible loss of fidelity.

These approaches to delivery clearly demonstrate the potential for semantics to improve the process, and indeed merely scratch the surface in terms of the semantic

\textsuperscript{13}or audio, although silence is less prevalent there
concepts that may be useful in guiding delivery. For further examples, see Section 6.1 in Chapter 6.

2.4.5 Ontology-based Metadata

In a recent study, Nack et al [108, 109] compare the Semantic Web [110] and MPEG-7 [71] initiatives in the pursuit of an interoperable multimedia metadata framework for the web. They conclude that MPEG-7 is essentially monolithic—designed to be a single standard that everybody will use that should be able to do just about anything required of it. To wit, XML is used for its syntax and the semantics are hard coded in the standards documents, which hinders semantic interoperability between MPEG-7 and other domains.

In contrast, the Semantic Web is specifically designed for semantic interoperability, providing the Resource Description Framework (RDF) [111], RDF-Schema (RDFS) [32] and OWL (Web Ontology Language) [33] for increasing levels of expression of semantic relationships in a machine-understandable format. This greatly simplifies the process of using semantic information from independent sources together in order to draw inferences from the information which would otherwise be unavailable.

While MPEG-7 does not facilitate interoperability, it provides a number of general features which are essential for multimedia metadata, including data types specific to audio and visual data, and spatial, temporal and spatio-temporal multimedia segmentation structures. Because of this, Hunter [112] has endeavoured to provide the
beginnings of an RDFS/DAML+OIL\textsuperscript{14} [113] model for MPEG-7 to provide these features while permitting interoperability. In a later paper [114], Hunter combines the MPEG-7 model with a high-level ontology in order to demonstrate the use of MPEG-7 semantics across domains—specifically in the description of Museum collections and Digital Libraries. In [114], Hunter also develops an ontology to represent the MPEG-21 Rights Data Dictionary (RDD) [115]—part of the Multimedia Framework for expressing permissions on Digital Items and relationships with the Users to which permissions are assigned.

RDF and OWL are used in Chapter 6 as one way to make cross-domain semantic reuse easier. The ontologies developed by Hunter can potentially be used to simplify integration of MPEG-7 or MPEG-21 RDD into the semantic delivery framework in that chapter.

\textbf{2.4.6 Summary}

Scalable content formats such as SVC and SLS provide a great deal of flexibility to the process of delivering content. They allow a single version of a piece of content to be stored, yet the most suitable resolution, frame-rate, and so on be delivered to each client. Content adaptation of this form is part of the vision of Universal Multimedia Access (UMA), which is discussed in Chapter 3.

Scalable content formats also allow much finer-grained control over the optimisa-

\textsuperscript{14}DAML: the DARPA Agency Markup Language; OIL: Ontology Inference Language. The combination of these two languages were the forerunner to OWL.
tion process involved in delivering content over a channel that in a dynamic sense has insufficient bandwidth to deliver the entire data. In this case scalable content allows much more graceful (or often imperceptible) degradation of the received media. There are numerous examples of using content semantics to tailor this process even further. Chapter 6 presents a generic framework for this process of incorporating semantics into delivery optimisation.

2.5 Conclusion

Freeing multimedia from hard-coded formats requires, first of all, a way to describe individual formats that is different to the traditional method of hard-coding into software or hardware. The first part of this task is to describe the syntax of a format, and doing this enables numerous processes—format-independent content streaming and delivery (see Chapters 4 and 5), and mapping from a format to format-independent metadata for RDO (Chapter 6). There are numerous syntax description languages tailored to specific domains; for Multimedia this includes XFlavor, BSDL, and a combination of the two—BFlavor. In addition to the static specification of syntactical constructs provided by all meta-syntaxes, these latter languages also describe how to dynamically parse a given instance of a syntax. BSDL is an extension of XML Schema, and as a result provides an elegant way to use an XML-based expression (for example an XPath expression) to identify binary content. For this reason it is used as the language for syntax description throughout this thesis. Prior to this, Chapter 3 proposes ways to address some shortcomings of the language.
The first application to which BSDL is put is format-independent multimedia streaming and delivery (Figure 2.19). This application builds on hinted streaming servers, which perform the majority of the work in preparing media for delivery offline, providing a sequence of format-independent byte-copy instructions for the streaming server to perform. Chapter 4 proposes a means to make both the server and the hinter format-independent, and Chapter 5 uses this system to develop the format-independent Delivery Node.

Chapter 6 looks up the hierarchy in Figure 1.2 and aims to link higher levels with the multimedia delivery process. This process draws on scalable content formats, adaptation, and Rate-Distortion Optimisation, and ontological languages to incorporate any semantics that are useful in content delivery (whether at an initial delivery node or within the network—such as a gateway processing node—Figure 2.19).

An understanding of syntax alone is insufficient to perform many operations on multimedia content, such as decoding (at the content consumer—Figure 2.19). The work in Appendix A is part of the Reconfigurable Media Coding (RMC) project being undertaken by MPEG [116]. RMC provides a way to make multimedia content “self-describing”, in that metadata is carried along with the content that is sufficient
to allow a format-independent RMC decoder to decode it. There are two parts to this metadata: a description of how to build the decoder from fundamental building blocks, and a description of the syntax of the content itself. Appendix A proposes a mechanism for using BSDL to provide this syntax description, as well as a proposal for deriving a parser module from this description.
Chapter 3

Laying the foundations

A universal metasyntax for multimedia

“What’s this fish doing in my ear?” [asked Arthur.]

“It’s translating for you,” [Ford replied.] “It’s a Babel fish. Look it up in the book if you like.” ...

“The Babel fish,” said The Hitchhiker’s Guide to the Galaxy quietly, “is small, yellow and leech-like, and probably the oddest thing in the Universe. It feeds on brainwave energy received not from its own carrier but from those around it. It absorbs all unconscious mental frequencies from this brainwave energy to nourish itself with. It then excretes into the mind of its carrier a telepathic matrix formed by combining the conscious thought frequencies with nerve signals picked up from the speech centres of the brain which has supplied them. The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language.”

— The Hitchhikers Guide to the Galaxy, 1979, Douglas Adams
Researchers have yet to come up with anything quite as ...err... ergonomic, as the Babel fish. There is, however, substantial literature on the subject of Universal Multimedia Access (UMA), the idea of being able to access any multimedia content, anywhere, and at any time [117]. Much of this effort has been aimed at devising means to interact with multimedia content—store, deliver, process, consume it, and so on—in ways that are independent of the particular representation in which the content is embodied. This format-independent approach to multimedia is increasingly important, as more and more encoding formats are developed, and the diversity of multimedia devices and networks continues to grow [10].

This chapter focuses on one of the foundations of the UMA approach: a meta-language for describing the structure of multimedia data. The meta-language is known as the Bitstream Syntax Description Language (BSDL) [28], and it allows a device to perform operations on data without requiring specific software for every format it may encounter. This is in contrast to the current status quo, where the release of a new multimedia content format (H.264/AVC [12], to cite a recent example), requires providers of multimedia software and hardware to (largely independently) develop new modules for their products in order to take advantage of the format.

BSDL was originally developed as part of the Adaptation component of the MPEG-21 Multimedia Framework. In this context, it was used to describe the macro-level structure of scalable content, so that temporal, spatial, or SNR adaptation might be implemented by identifying which portions of the content to transmit, and which
to drop [118]. Consequently, once a target frame rate, resolution, or quality level is identified, the actual adaptation is executed by a processor that knows nothing about the bitstream beyond what it is given by BSDL. This adaptation can occur on a streaming server, a gateway node, or on the consuming device (Figure 3.1).

Beyond scalable content adaptation, the format-independence provided by BSDL is central to other applications that help to fulfil the UMA vision. These applications are described in other chapters of this thesis, but a brief outline is given here, with reference to Figure 3.1.

- A format-independent framework for streaming (and otherwise delivering) multimedia content (Chapter 4 on page 95) uses BSDL as an abstraction layer for binary media so that it can be easily manipulated with an XML syntax. The framework enables Streaming Servers to deploy content in new formats as they are developed, without the need for additional software;
The framework for incorporating semantic information in multimedia delivery presented in Chapter 6 on page 153 uses BSDL to extract metadata from binary content to be used by the delivery process (such as timing information), as well as to link semantics to fragments of content;

Finally, Appendix A presents a reconfigurable decoder architecture that allows both Special- and General-purpose multimedia Devices to render content in any format, without prior software support. The syntax of such content is described by BSDL, providing the flexibility to alter the structure of the bitstream as desired.

The remainder of this chapter is concerned with the identification and rectification of several shortcomings with BSDL that become evident when it is used to describe recent coding formats such as H.264/AVC [12], Advanced Audio Coding (AAC) [119], or Scalable-to-Lossless (SLS) audio coding [120]. This analysis is presented in Section 3.2 on page 72. Prior to this, Section 3.1 on the facing page provides a brief tutorial on the features of BSDL that are pertinent to the subsequent discussion. This tutorial expands on the example presented earlier in Section 2.1.1 and Figure 2.2d on page 23. Section 3.3 on page 81 proposes several extensions to the language that address the identified issues, and 3.4 on page 90 presents empirical data justifying the proposals.

These extensions have subsequently been incorporated into the language [40]. Furthermore, the new applications of BSDL proposed by this thesis have led to its separation from MPEG-21 into a first-class standard in its own right [121].
3.1 A brief tutorial on BSDL

BSDL provides a description of the structure of multimedia data using XML. Listing 3.1 gives an example BSDL Description, for a video in the H.264/AVC coding format. In general, multimedia data may be described as a sequence of binary symbols of arbitrary length—some symbols containing a single bit, while others many bytes. The BSDL Description indicates the value of these binary symbols in a format that is both human- and machine-readable, for example using hexadecimal values (as is the case for startCode, in Listing 3.1) or integers or strings. It also organises the symbols into a hierarchical structure that reflects the semantic interpretation of the data.

Multimedia bitstreams typically contain millions or billions of symbols, and for many of the applications of BSDL it is neither necessary or efficient to describe every single one. Instead, BSDL makes it possible to describe large segments of data using pointer-like structures, indicating the offset and length of the segment in the raw content (known as a byteRange). In this way, unnecessary detail can be hidden from an application, by encapsulation within an opaque structure. In Listing 3.1, the payload field is a pointer to a byteRange beginning at byte 5 and continuing for 100 bytes.

In other words, the level of detail of the BSDL Description may be fully customised to the requirements of the application: for the same multimedia data, one application may use BSDL that describes some or all of the fields that another hides within a
**byteRange.** As a result, the BSDL Description does not replace the original data, but instead provides additional information (or metadata) to help an application to parse and to process the content. Finally, it should be noted that the names of the elements in the BSDL Description are not mandated by BSDL; it is the choice of the application to assign names that provide meaningful semantics for the description at hand.

In the example (Listing 3.1), we can see that BSDL describes the values of binary symbols within the multimedia data. However, by itself, this description lacks certain details that are fundamental to an application that needs to process the data. Specifically, while the example describes one particular bitstream, it does not provide any information about how a similar description may be derived for a different bitstream encoded using the same format. Also, if the aim is to reconstruct the binary data from a processed description, further information is missing, such as the number of bits to use when writing each symbol.

Because all multimedia data in a given format conforms to a single (although potentially flexible) structure, this structural information is expressed, not in the BSDL Description itself, but in a separate Schema. This BSDL Schema specifies the encoding associated with all instances of data in the given format.

BSDL is an XML language, and so BSDL Schema builds on top of XML Schema [31], a W3C tool for expressing constraints on the structure and datatypes of a family of XML documents. BSDL Schema adds further constraints to its parent language that enable the Schema to elegantly describe not only the XML (the BSDL Descrip-
Listing 3.1: A BSDL Description exposes the fields in a multimedia bitstream, in this case a H.264/AVC video

With a BSDL Schema, it is therefore possible to parse a bitstream, generating a description that may be subsequently processed. Furthermore, as previously described in Section 2.4.2 on page 47, the BSDL Schema also enables an application to reconstruct a processed version of the multimedia data from its description.

BSDL is an extension of XML Schema [31]. It is the latter that defines the structural features of a BSDL Schema (known as particles): `<choice>`, `<sequence>`, `<all>`, `<element>` and `<group>`. Elements are given a `type`, which is defined by either a `<complexType>` (the element contains another particle), or a `<simpleType>` (the
Listing 3.2: A BSDL Schema describes the structure common to all H.264/AVC video bitstreams
element contains binary content). BSDL defines how simple types are read from the bitstream. For example, the facet <maxExclusive> defines the number of bits read by integer types, and <length> the number of bytes in a string or hexBinary type. BSDL also adds annotations that provide the additional information required for bitstream parsing, including various means to decide which choice option is actually chosen, whether optional particles are in fact present, and how many occurrences of multiple particles exist. Listing 3.2 on the preceding page shows one example of these annotations: the bs2:ifNext attribute (at marker (a)). This attribute specifies that the structure to which it is attached should be read from the bitstream only if the next bytes in the bitstream correspond to the hex value of the attribute\(^1\). Other pertinent features of the language are:

(b) the element named startCode in Listing 3.1 is not to be confused with the BSDL Schema pseudofacet\(^2\) bs2:startCode (Listing 3.2(b)). The latter specifies a value that indicates the start of the subsequent element. The data will be read into the current element until the value is found, or the end of the stream is reached. A similar pseudofacet—bs2:endCode provides a value that represents the end of the current element.

(c) bs2:if (Listing 3.2(c)) is similar to bs2:ifNext except that it evaluates a boolean expression rather than a value in the raw bitstream.

(d) bs2:nOccurs specifies the number of occurrences of the pic_scaling_list ele-

\(^1\)bs2:ifNext has other options too; the interested reader is directed to [40].
\(^2\)Termed pseudofacet because strictly speaking XML Schema specifies a closed set of facets [40]. As a result, pseudofacets must be enclosed in the XML Schema extension elements xsd:annotation/xsd:appinfo.
ment (Listing 3.2(d)).

3.2 An analysis of BSDL with recent coding formats

This section identifies the issues associated with the Bitstream Syntax Description Language using the recent H.264/AVC, AAC, and SLS encoding formats as examples. Of primary concern are the extensive use of entropy-coded values in header fields, parameter sets, and dynamic length bit-fields.

3.2.1 Entropy-coded header fields

H.264/AVC makes extensive use of the Exponential-Golomb (expGolomb) encoding [12] throughout the header fields of most bitstream structures. For example, the ID fields shown in Figure 3.2 on page 75 are represented by expGolomb codes. Parsing the Bitstream for adaptation requires that the Syntax Description tool be capable of interpreting such codes. However, BSDL [118] currently supports only simple numerical types and strings.

It must be noted that the seminal work on BSDL states that

“BSDL does not and cannot have the ambition to parse any bitstream at any level of detail. . . . In particular, the main part of a bitstream—qualified hereafter as the ‘payload’—is usually the output of an encoding process such as entropy coding, wavelet or Discrete Cosine Transform.”
Such sequence of bits can only be decoded via an algorithm implemented in a programming language. It is obviously not the ambition of BSDL or Flavor to natively decode such data [29].”

However, as in the case of expGolomb, such codes are no longer solely used in the payload. Therefore, if it is the ambition of BSDL to parse H.264/AVC at any level of detail at all, it must support expGolomb-coded data types. For this reason, the specification is currently being amended to normatively specify an expGolomb data type [122]. This solution, while satisfactory for expGolomb, is very inflexible. It effectively necessitates amendments to the standard (a slow process, at best) for every new encoded data type to be processed by BSDL in the future.

Another limitation of BSDL cited in [29] is the “the inherent verbosity of XML”. Devillers states that

“BSDL was primarily introduced for the problem of content adaptation, and thus is meant to describe bitstreams mainly at a high syntactical level, which should limit the verbosity of the descriptions.”

Indeed, de Neve et al. [42] report that even their high-level description of H.264/AVC is five-and-a-half times (550%) the size of the bitstream! However, this (significant) limitation is obviated if the XML description is not produced in the first place. Instead of generating a verbose XML description of the bitstream, the BSDL Schema may be used to interact directly with it directly (in the binary domain). This is discussed further in Chapter 4 and Appendix A.
3.2.2 Parameter Sets

Many of the fields found in packet headers of previous video formats tend to have identical values across numerous packets. H.264/AVC introduces the concept of a Parameter Set, which extracts much of this data into a separate structure. The Audio Data Interchange Format (ADIF) of AAC [119] is another example of the use of such parameter sets.

Parameter sets can be sent less frequently and potentially with a more reliable mechanism, since the perceptual quality of the video output is generally highly dependent upon such information [123]. Two levels of parameter set are used: the Picture Parameter Set (PPS), which provides parameters common to groups of pictures, and the Sequence Parameter Set (SPS), which specifies parameters common to groups of picture sequences.

Both a PPS and the SPS it references contain data required to parse each and every picture element (slice). For example, the SPS contains a field specifying how many bits are used for the frameNum field in a slice header. To retrieve this information, the slice header has a pointer to a PPS (ppsID—Figure 3.2 on the next page), which in turn points to the SPS. Both SPS’ and PPS’ may evolve over time—subsequent parameter sets may replace earlier ones with the same ID; for any slice the correct SPS/PPS are the most recently received SPS and PPS with matching IDs.

When parsing a H.264 bitstream for adaptation, a BSDL parser must perform this dereferencing operation many times per slice. The MPEG conformance schema for
H.264/AVC [124] uses the following XPath expression to dereference sequence parameter sets:

\[
/\text{avc:bitstream}/\text{avc:byte}\_\text{stream}/\text{avc:byte}\_\text{stream}\_\text{nal}\_\text{unit}[\text{avc:nal}\_\text{unit}/\text{avc:raw}\_\text{byte}\_\text{sequence}\_\text{payload}/\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{rbsp}/\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{id} = /\text{avc:bitstream}/\text{avc:byte}\_\text{stream}/\text{avc:byte}\_\text{stream}\_\text{nal}\_\text{unit}[\text{avc:nal}\_\text{unit}/\text{avc:raw}\_\text{byte}\_\text{sequence}\_\text{payload}/\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{rbsp}/\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{id} = /\text{avc:bitstream}/\text{avc:byte}\_\text{stream}/\text{avc:byte}\_\text{stream}\_\text{nal}\_\text{unit}[\text{position()} = \text{last()}]/\text{avc:nal}\_\text{unit}/\text{avc:raw}\_\text{byte}\_\text{sequence}\_\text{payload}/\text{child:*}/\text{child:*}/\text{avc:slice}\_\text{header}/\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{id}[\text{position()} = \text{last()}]/\text{avc:nal}\_\text{unit}/\text{avc:raw}\_\text{byte}\_\text{sequence}\_\text{payload}/\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{rbsp}/\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{id}[\text{position()} = \text{last()}]/\text{avc:nal}\_\text{unit}/\text{avc:raw}\_\text{byte}\_\text{sequence}\_\text{payload}/\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{rbsp} \ldots (3.3)
\]

This schema declares an element for every last item in the AVC pseudocode, even for things that are not identifiers, such as if and for expressions. In many cases this does not add any meaningful structure to the BSDL, yet it clearly greatly adds to the complexity. Debugging the above XPath expression would be a nightmare! After removing the redundant elements of Listing 3.3, the expression is somewhat less verbose, although still relatively complex:

\[
/\text{avc:h264}/\text{avc:sequenceParameterSet}[\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{id} = /\text{avc:h264}/\text{avc:pictureParameterSet}[\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{id} = /\text{avc:h264}/\text{avc:slice}[\text{last()}]/\text{avc:pic}\_\text{parameter}\_\text{set}\_\text{id}][\text{last()}]/\text{avc:seq}\_\text{parameter}\_\text{set}\_\text{id}][\text{last()}] \ldots (3.4)
\]
Even worse than the maintainability problems of this expression is the fact that it requires $O(n^2)$ time to process (where $n$ is the number of NALUs in the stream). This is because the predicate expression [last()] must iterate through the entire list of NALUs to that point to verify which one is in fact the last instance of the particular type: an arithmetic progression with a total of $n(n + 1)/2$ comparisons.

An alternative approach could be to iterate backward from the slice being processed. In this way, the first SPS and PPS encountered that have the correct ID would be the valid choices. Unfortunately, inside the predicate where the PPS is referenced, XPath does not provide any way to access the slice being processed (the context at the beginning of the expression. Interestingly, XSLT [47] provides the current() function for precisely this purpose. Using this function the expression would be a little simpler:

$$\text{preceding::avc:sequenceParameterSet[avc:seq\_parameter\_set\_id = current()]/preceding::avc:pictureParameterSet[avc:pic\_parameter\_set\_id = current()]/avc:pic\_parameter\_set\_id[1]/avc:seq\_parameter\_set\_id[1]} \ldots (3.5)$$

Unfortunately, current() is not available to XPath, and in any case, Listing 3.5 is still $O(n^2)$ because preceding:: must also iterate through the sequence of NALUs. The complexity of Listing 3.4 is shown empirically in a later section (Figure 3.4 on page 90), but to verify the assertion that the cause is indeed [last()] and preceding::, Listing 3.6 and 3.7 on page 78 show test schemata that isolate the offending expressions. The run-time of each schema is measured using the BSDL reference implementation [125] with two widely used XPath implementations\(^3\), over a range of

\(^3\)Saxon 8.9B [126] and Apache Xalan 2.7.0 [50]
values for /test/element[1] (see Figures 3.3 on the next page).

While the absolute efficiency varies widely between the two XPath implementations, all display $O(n^2)$ time complexity. For the [last()]) predicate, it is possible that a more advanced implementation could keep an index of elements and be able to determine the last (or indeed $n$th) element of any given name in constant time. However, this is not possible for any arbitrary predicate, and in particular it is unlikely that any processor would index a-priori the predicate [avc:pic._parameter_set_id = current()/avc:pic._parameter_set_id][1]/avc:seq._parameter_set_id] (not to mention the larger predicate of which it is a part—Listing 3.5 on the facing page). BSDL is intended to be used on a wide range of platforms, so it cannot rely on implementations to eliminate what would otherwise be quadratic-time expressions. As it turns out, it is possible for BSDL to implement such indexing operations itself, by providing a syntax and semantics for variable declaration (discussed below—Section 3.3.2 on page 86).
Listing 3.6: BSDL test schema for 
[last()] 

Listing 3.7: BSDL test schema for preceding:: 

Figure 3.3: BSDL Schemata and test results demonstrating quadratic-time performance of the SPS dereference operation.
3.2.3 Dynamic length bit-fields

Another common feature in H.264/AVC is header fields which use a variable number of bits. Some (such as those with expGolomb encoding) may be parsed without external information. For others, the length of the field is indicated elsewhere within the bitstream. The frameNum field in Figure 3.2 on page 75 is one such example. Another is the maximum bit-plane side-information in the SLS scalable audio coder [120].

BSDL provides a mechanism for resolving XML Schema `<union>` datatypes, which allows fields with variable length to be described (bs2:ifUnion—Listing 3.8 on the next page). However, this necessitates an XPath expression for each possibility within the union. When the location of the length data is specified in a SPS or PPS (as is in fact usually the case for H.264), the computation complexity of the BSDL Schema increases to $O(pn^2)$.

Additionally, every possible bit-length must be individually enumerated. Even for the given example this is particularly error prone; a field with possible lengths from 1 to 32 bits will quickly become untenable. A more compact approach is proposed in Section 3.3.3.

\[^{4}p\text{ is the number of }\text{ifUnion}\text{ elements evaluated before a true result is returned}\]
Listing 3.8: BSDL schema fragment showing union data types—excerpt from the MPEG conformance schema for H.264/AVC [124]
3.3 Extensions to BSDL to allow its use with recent content formats

Description of recent content formats using BSDL in its present state, while theoretically possible, is implausible for any bitstreams of real-world length, due to the scalability issues discussed in Section 3.2 on page 72, and as demonstrated in Section 3.4 on page 90. However, it is possible to address these issues with a number of extensions that exploit currently unused facets of XPath. These are a means to create new encoded data types (Section 3.3.1), dynamic bit-length types (3.3.3 on page 88), and XPath variable declarations (3.3.2 on page 86).

3.3.1 ECMAScript implementation of extension datatypes

Rather than specify encoded data types as part of the BSDL standard, this section proposes an addition to BSDL to allow custom parsing operations to be specified within the BSDL Schema using ECMAScript [127]. An example of this is shown in Listing 3.9 on page 84, where a user-defined data type has a bs1:codec attribute that references its implementation. The implementation is provided by a bs1:script node, which is required to specify two functions—BintoBSD() for the binary to BSD process, and BSDtoBin(value) for the reverse. BSDL provides read(nBits) and write(value, nBits) functions to allow the user functions to access the bitstream. The example in Listing 3.9 on page 84 implements the expGolomb parsing process specified by [12].
Laying the foundations

Data types referenced by `bs1:codec` (Listing 3.9(b)) in a BSDL Schema may be implemented using ECMAScript and the implementation embedded in the BSDL Schema via the `bs1:script` component (Listing 3.9(a)). This allows arbitrary parsing algorithms to be specified by a BSDL Schema for use by BintoBSD and BSDtoBin parsers, enabling the processing of data structures that cannot be specified using other BSDL syntax elements. The `bs1:script` component defines the local name of the datatype, which inherits the target namespace of the schema document. The `bs1:codec` attribute can then reference this implementation via the URI of the datatype, which is obtained by appending the local name as fragment identifier to the namespace URI.

As another example, ECMAScript datatypes may be used to allow a BSDL Parser to process Variable Length Codes, such as Huffman codes or Arithmetic-coded values. An ECMAScript implementation may be referenced by `bs1:codec` in the following ways:

- The value of `bs1:codec` is a URL that resolves to a BSDL Schema, with a fragment identifier corresponding to the value of an id attribute on a `bs1:script` element;

- The value of `bs1:codec` is a URL that resolves to an ECMAScript file, with a fragment identifier corresponding to the name of a class within that file; or

- The value of `bs1:codec` is a URL that resolves to an ECMAScript file, with no fragment identifier.
In each case, a BSDtoBin parser shall search the `bs1:script` element, class or file (respectively) for a function (or method) with the signature BSDtoBin(value). The BSDtoBin parser shall call this function to parse the element to which `bs1:codec` is attached. The BSDtoBin parser shall pass as `value` the text content of the element if the content is simple, or the element and its descendents, otherwise.

A BintoBSD parser shall search the `bs1:script` element, class or file (respectively) for a function (or method) with the signature `BintoBSD()`. The parser shall call this function to generate the element to which `bs1:codec` is attached. The `BintoBSD()` function should return either a string containing the lexical value of the element, or a DOM Element representing the element.

`BintoBSD()` and `BSDtoBin()` may call the functions specified in Table 3.1 on page 85 to access data from the BintoBSD and BSDtoBin parsers. `BintoBSD()` and `BSDtoBin()` may also instantiate Objects specified by the ECMAScript bindings for the Document Object Model: Level 3 Core Specification [128].

Finally, the functions declared by `bs1:script` may also be invoked from within XPath expressions, in the same manner that built-in XPath functions are called. This provides a mechanism for XPath expressions to return the result of computation that is more complex than is easily expressed in XPath.
Listing 3.9: BSDL Schema for the Exp. Golomb as defined by H.264
Table 3.1: BSDL-defined ECMAScript functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read(bits)</strong></td>
<td>This function shall be provided by a BintoBSD parser and may be called by the BintoBSD() function of a bs1:script component. When this function is called, a BintoBSD shall read from the bit-stream the number of bits specified by the integer value of the bits parameter, and return the unsigned integer value of the bits read.</td>
</tr>
<tr>
<td><strong>write(value,bits)</strong></td>
<td>This function shall be provided by a BSDtoBin parser and may be called by the BSDtoBin(value) function of a bs1:script component. When this function is called, a BSDtoBin parser shall write to the bitstream the unsigned integer value of the value parameter, using the number of bits specified by the integer value of the bits parameter.</td>
</tr>
<tr>
<td><strong>xpath(exp,type)</strong></td>
<td>This function shall be provided by a BintoBSD parser and may be called by the BintoBSD() function of a bs1:script component. When this function is called, a BintoBSD parser shall execute the XPath expression declared by the string value of the exp parameter, and return the value of the result of the expression. The expression shall be evaluated in the context of the partially instantiated BSD. The type of object returned by the function shall be specified by the value of the type parameter, and shall be one of the following symbolic constants:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODESET</td>
<td>A NodeList object.</td>
</tr>
<tr>
<td>NODE</td>
<td>A Node object</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>A Boolean primitive</td>
</tr>
<tr>
<td>STRING</td>
<td>A String primitive</td>
</tr>
<tr>
<td>NUMBER</td>
<td>A Number primitive</td>
</tr>
</tbody>
</table>

NOTE The NodeList and Node types are defined by the Document Object Model, Level 3: Core [128], and the Boolean, String and Number primitive types are defined by the ECMAScript specification.
### 3.3.2 XPath Variable declarations

The complexity of the dereferencing operation presented in Figure 3.2 on page 75 may be greatly simplified by recognising that the set of valid PPS (and equally SPS) at any point in time may be represented as an object array, indexed by the respective ID. Any time a new parameter set is encountered in the bitstream, it is inserted into the array in the appropriate location, replacing any parameter set that was previously in that slot. XPath does not directly support an array data type, however it does provide node-sets, which may be used for this purpose. In order to store a node-set of all the PPS (or SPS), the node set must be registered as an XPath variable. In so doing, the expression to locate the relevant SPS for a given slice simplifies to

\[
\text{sps} \left[ \text{pps} \left[ \text{avc:ppsID+1} \right] / \text{avc:spsID+1} \right]
\]

... (3.10)

This is a significant result, reducing the complexity of listings 3.3 and 3.4 to two arrays and two indices. Furthermore, since this expression doesn’t involve any iteration over the set of slices, its evaluation time is \(O(1)\), and the \(O(n^2)\) complexity observed in the base specification consequently reduces to \(O(n)\). In order to enable the use of arbitrarily defined XPath variables in BSDL, a mechanism must be provided to assign their value, and also their position within the node-set. The \text{bs2:variable} element is proposed for this purpose, as described below. An example is shown in Listing 3.11 on page 88.

The addition of \text{bs2:variable} allows a BSDL Schema to declare variables that may be subsequently referenced from within XPath expressions. In particular, variables
may be used to simplify XPath expressions by replacing Node-tests with variable references. This can enhance the readability and execution speed of the expression. This component may be placed as the child of a `xsd:annotation/xsd:appinfo` combination under an `xsd:element` component.

The partial-value of a variable shall be one or more of the following:

- If the `position` attribute is present, a Node-set;
- If the `value` attribute is present, the result of evaluating the `value` expression; and/or
- The current instantiation of the element node on which the variable is declared, otherwise.

The `name` attribute indicates the local name of the variable. The `value` attribute contains an XPath expression which shall be evaluated against the partially instantiated BSD, relative to the current instantiation of the enclosing element. The result of the XPath expression shall be stored as the partial-value of the variable being declared. If the `value` attribute is not present, then the partial-value of the variable shall be the element node being instantiated. The `position` attribute contains an XPath expression which shall be evaluated against the partially instantiated BSD, relative to the current instantiation of the enclosing element. The XPath expression shall resolve to a strictly positive integer.

NOTE In other words, the first position within a variable node-set is 1.
Laying the foundations

When the position attribute is present, the value of the variable shall be a node-set. The partial-value computed above shall be stored in the position indicated by the result of the position expression. If the partial-value is a node-set with more than one item, then the items in the partial-value are stored within the variable node-set in consecutively increasing positions, beginning with the result of the position expression. If the position attribute is not present, then the value of the variable shall be the partial-value computed above.

3.3.3 Dynamic bit-length simple types

The use of xsd:union to specify variable length fields such as frameNum, while possible, is not intuitive and prone to error. A far simpler solution is to provide a facet which directly calculates the length (in bits) of the type being defined. Indeed, such a facet exists within BSDL to specify a dynamic length in bytes—bs2:length; it is proposed to introduce a similar facet specifying bit-length. This has the effect of replacing numerous XPath expressions (12 in the example—Listing 3.8 on page 80) with a single expression, reducing verbosity by 97.5% in the process (and hence considerably increasing maintainability).

Whereas the <bs2:length> facet constrains the size in bytes of datatypes such as
Listing 3.12: BSDL Schema fragment describing the frameNum field of an H.264/AVC bitstream

bs1:byteRange or xsd:hexBinary, the <bs2:bitLength> facet constrains the size in bits of an unsigned integer. In other words, it can describe an unsigned integer that is encoded on a dynamic number of bits. bs2:bitLength may be used to restrict the xsd:unsignedLong, xsd:unsignedInt, xsd:unsignedShort or xsd:unsignedByte data-types. The value of this facet shall be interpreted as an XPath expression, resolved against the partially instantiated BSD, with the context node set to the element being instantiated. The equivalent integer value of the expression shall be used as the length in bits of the data type being restricted.

When instantiating an element that is restricted by bs2:bitLength a BintoBSD parser shall add an xsi:type attribute to the element, with a value corresponding to one of bs1:b1 … bs1:b32 according to the number of bits assigned to the element5. The behaviour of BintoBSD is not specified if the resulting value is greater than 32.

5BSDtoBin is agnostic about the BSDL-2 extensions and therefore ignores the bs2:bitLength facet. This is why the number of encoding bits has to be specified with the xsi:type attribute and bs1:b1 … bs1:b32.
3.4 Results

Each of the proposed solutions were tested via modifications to the BSDL reference software [125], on H.264/AVC bitstreams of varying length. Tests were carried out on a P4 1.7Ghz machine with 512MB of RAM, running Linux and the Sun JVM 1.5.0. Each trial was repeated three to ten times, and the results averaged (in each case the result of the first run was excluded because it includes additional time for JVM initialisation and class loading).

As can be seen, the time required to process the bitstream using the unmodified BinToBSD software (the base series in Figure 3.4) increases according to $O(n^2)$—for a two minute bitstream, BinToBSD processing time exceeds an hour.
Subsequent series’ in Figure 3.4 show the performance of the various proposed solutions. The introduction of user-defined types (Section 3.3.1 on page 81) is not intended to improve the scalability of the system. Rather, it enhances extensibility, by simplifying adoption of new encoded data types, and results demonstrate that the overhead of this enhancement is negligible.

The Bit-Length Pseudofacet (Section 3.3.3) reduces parse time for this bitstream by approximately 75%, because of the removal of the $p$ term discussed in Section 3.2.3 on page 79. Although this considerably simplifies the Schema, complexity is still $O(n^2)$; the system still cannot scale to video of real-world duration. In contrast, the introduction of user-defined variables (Section 3.3.2 on page 86) reduces the complexity to $O(n)$. This results in a Bitstream Syntax Description Language that may be used on real-world H.264/AVC streams; the series’ labeled all in Figure 3.4 and 3.5 show the performance of BinToBSD with all modifications applied. The latter verifies the $O(n)$ performance of the algorithm for bitstreams in excess of an hour. The resulting BSDL processing is around 10 times faster than playback (on the test machine).

A further benefit of the proposed modifications to BSDL is shown by the lowMemory series. All XPath expressions are now evaluated against the set of user variables, meaning that the BSD itself is no longer required to be stored in memory. This has the effect of removing the correlation between bitstream duration and memory consumption, now constant at approximately 5 MB.

\footnote{Data tables for these graphs are available in Appendix B.}
3.5 Conclusions

This chapter has demonstrated that the Bitstream Syntax Description Language (BSDL) displays $O(n^2)$ complexity when processing modern content formats such as H.264/AVC, and is consequently unable to process such bitstreams of real-world duration. It has proposed a number of additions to BSDL which have the effect of reducing the complexity of BinToBSD processing to $O(n)$, as well as improving its
efficiency, maintainability, and extensibility:

- a means to specify new encoded data types within a BSDL Schema (Section 3.3.1). This considerably improves the flexibility and extensibility of BSDL over the existing approach, which was to specify new encoded data types within the BSDL specification;

- XPath variable declarations (Section 3.3.2). This greatly reduces the complexity of the parameter set dereferencing operation (very common in H.264/AVC), as well as many others. It also fundamentally reduces the complexity of this operation from $O(n^2)$ to $O(n)$; and

- dynamic bit-length types (Section 3.3.3). These can be used to replace union types (which in BSDL iteratively compute $p$ XPath expressions until one evaluates to true). In contrast, dynamic bit-length types compute a single XPath expression to evaluate the length of the field, resulting in a $p$-fold reduction it processing time. Furthermore, this proposal greatly improves maintainability, markedly reducing the verbosity of specifying such a type—in the case of frameNumType (see Section 3.3.3) by 97.5%.

These results have been validated with bitstreams up to an hour in length. Appendix H and Listing H.1 show the complete BSDL Schema for H.264/AVC incorporating the amendments proposed in this chapter. This Schema is used by the work in Chapter 5 as an example for the format-independent delivery framework.

---

7 a slow process, at best!
Chapter 4

Babel’s formwork

Format-independent streaming and delivery

The amount of multimedia content available via the Internet, and the number of formats in which it is encoded, stored and delivered continues to grow rapidly. So too the number and diversity of the devices and software applications which produce, process and consume such content [10]. This constantly changing landscape presents an increasing challenge to interoperability, since more and more software and hardware must be upgraded as new formats are developed. However, many of the operations performed on multimedia content are similar across coding formats. Consequently, numerous approaches have been developed that process content using generic software, with format-specific details provided in a data file. This considerably simplifies interoperability, since support for a new content format may be provided by disseminating a simple file, rather than requiring application providers to extend and modify their software. Examples of this approach are Flavor [30], which is a C++-like syntax that describes the structure of multimedia content to automate parser generation, and the
recent MPEG Reconfigurable Media Coding project [116] (see also Appendix A), which targets the provision of a declarative language for building entire codecs by combining primitive function blocks.

There have been a number of approaches developed in recent years that are designed to facilitate transaction and processing of rich media\(^1\), including MPEG-21 [129] and TVAnytime [130]. These technologies are also—to greater and lesser degrees—format-independent in terms of the media and metadata components which make up objects within the respective approach. However, while they provide numerous tools for interacting with rich media objects, a fully format-independent framework for streaming and delivery of such objects is not available. Such technologies provide various virtual containers for multimedia content and metadata, but do not provide delivery mechanisms for such containers over the numerous channels on which users will want to access the content.

Furthermore, the format-independent nature of such containers means that it is not possible to provide a single mapping into a particular delivery format (such as the Real-time Transport Protocol (RTP) [131]). Mappings may exist for some content types that could be placed in the container, MPEG-1/-2 [25], or Digital Item Adaptation metadata [132] for example. However, in general, the existence of the required mapping for every potential component cannot be relied upon. A solution is therefore required which can specify any and all required mappings between the many formats in which the content is stored and the many channels on which it is to be delivered.

---
\(^1\)that is, Multimedia Content augmented with metadata to enhance the experience of users interacting with the content
These mappings must be specified in a manner that can be machine-processed to actually perform the delivery. As with Reconfigurable Video Coding and Flavor, this approach would allow new formats to be supported without requiring any additional software to be written.

Chapter 2 described technologies that exist for some components of this task, but a solution for the problem as a whole is lacking. This chapter proposes the Bitstream Binding Language (BBL), which is a high-level declarative language that specifies how multimedia content and metadata is to be bound into a delivery format. Following its development by the author, BBL was adopted by MPEG as Part 18 of the MPEG-21 Multimedia Framework (Digital Item Streaming) [73].

Application scenarios for this technology are covered in more detail below (Section 4.1.1), but it is important to point out that BBL is not a new delivery format. BBL does not replace RTP, or any other protocol or file format. Instead, it provides metadata information that describes to a streaming server (or other application, see 4.1.1) how to deliver content.

Given the application scenarios, Section 4.1.2 goes on to identify requirements for format-independent rich media delivery. A model for the delivery framework is then developed in 4.1.3. Section 4.2 on page 107 presents BBL in action, showing how it implements the model, and Section 4.3 on page 120 gives some prototypical computational complexity data and further discussion.
4.1 Developing A Framework For Format-Independent Multimedia Delivery

This section presents the development of a format-independent framework for rich media delivery. Part 4.1.1 introduces some relevant application scenarios, including multi-channel delivery, distributed content adaptation, generic syntax translation, and virtual container assembly. 4.1.2 identifies several requirements for the framework, from which an abstract model for the description of rich media delivery is developed (Section 4.1.3).

4.1.1 Application Scenarios

BBL was designed on the basis of a number of use-cases, some of which were identified by the MPEG-21 call for proposals on Digital Item Streaming [133]. The following sections presents these use-cases in abstract terms. Chapter 5 on page 127 details how BBL may address each case.

4.1.1.1 A format-independent streaming server [133]

Rich multimedia content may often be transmitted over several channels. For example, a media provider may wish to distribute their content to some users who connect via a PC and the wired internet, others on cellular smart-phones, and further users with a digital television receiver. Currently, the content provider would often store a separate version of the content for each usage scenario, and use specialised software
There is a many-to-many relationship between the formats in which content is available and the channels across which users wish to consume them. However, there is in general a many-to-many correspondence between the formats in which content is available, and the channels across which they are accessed (Figure 4.1). As a result, storing multiple versions of content quickly becomes untenable. Instead, using a format-independent approach, multiple bindings of the content can be provided as inputs to a generic processor. Each binding specifies how to adapt the content and metadata to the particular characteristics of the channel and/or receiving device, and describes how to map (and if necessary, transcode) the data into each output.

4.1.1.2 Generic multimedia syntax translation

As new content formats are developed, they do not become useful until infrastructure is deployed to allow the format to be processed and consumed. Additionally, the
increasing diversity of content and metadata specifications makes it difficult for—particularly portable—devices to interoperate with the breadth of content available. A format-independent binding language can be used as a lightweight mechanism to allow content to be converted from one format into another so that it may be consumed on legacy devices. For example, the Music Player Multimedia Application Format (Music Player MAF) [134] (Figure 4.2) is a new content format standardised by MPEG which combines MPEG-21 and MPEG-7 metadata, JPEG images and MP3 audio content in an ISO file format to provide an augmented digital music library. While existing devices—portable music players, mobile phones, PC software, and so on—may not support this format, a generic approach could be used to translate Music Player MAF content into representations that these devices understand.
Figure 4.3: A binding language could be used to assemble virtual containers

This makes it feasible for operators to begin to deploy content using new formats while still enabling it to be consumed by existing devices.

4.1.1.3 Virtual container assembly

A format-independent binding language may equally be used to describe the inverse process—the assembly of legacy content into new formats or containers such as MAFs or Digital Items. Figure 4.3 depicts this process: legacy content (for example MP3 audio with ID3 metadata) is processed by a generic processor [135] according to a description file, in order to generate a rich content container.

4.1.2 Requirements

These application scenarios highlight several requirements for a binding language:
4.1.2.1  Content format independence

A binding language must allow description of how multimedia content—image, audio, video, and so on—in any format is to be fragmented and packetised for delivery.

4.1.2.2  Metadata format independence

Most recent multimedia metadata specifications are specified with an XML syntax—for example MPEG-7 [136], Dublin Core [137], and TVAnytime [130]. It is likely that future metadata standards will increasingly be described using RDF, however it too is most commonly serialised in XML [111]. Given that the rich content containers discussed previously also use an XML syntax [130], native support for XML is clearly an important requirement. However, several widely used multimedia metadata technologies—notably ID3 [135] and EXIF [138]—use a binary syntax. This means that a format-independent binding language will need to be able to efficiently deal with metadata in these and potentially other arbitrary binary structures.

4.1.2.3  Delivery format independence

On the internet, RTP predominates multimedia transport. However, the application domains of the rich media standards discussed are considerably wider than this. Digital Television,—for example MPEG-2 Transport Streams over DVB-S/T/C [139]—Mobile Multimedia (3GPP [140] and DVB-H [139]), as well as digital radio (DAB/DMB [141])—are all potential channels for rich content delivery. A binding language should provide means to map content onto these and other delivery channels,
as well as provide means to extend the framework to new delivery channels as they are developed.

### 4.1.3 A model for generic delivery of rich media content

Figure 4.4 on the next page presents the model for multimedia content delivery which was developed for this work. It identifies the generic tasks associated with the streaming and delivery of multimedia content, independent of its format. As discussed (Chapter 2 on page 19), solutions for some parts of the model exist in the literature, and where appropriate they have been adopted into this work. However, there is a need for a solution which addresses the problem as a whole.

The task of multimedia delivery is considered as a series of smaller components: the extraction of fragments of one or more inputs, the combination these fragments into packets, and mapping of the packets onto output formats. Post-processing may also be required to compress or otherwise modify the data.
Figure 4.4: A model for multimedia content delivery
4.1.3.1 Binary Abstraction

Many recent multimedia metadata standards are expressed as XML. In fact, the hierarchical structure which XML provides is conceptually similar to the syntactical structures used in binary formats (as demonstrated by the BSDL mapping between binary and XML—see Sections 4.3.2 and A.6). Consequently, via a binary abstraction layer (Figure 4.4), XML tools can be used to manipulate binary data. This enables a single syntax for a format-independent binding language regardless of the format of the input data.

4.1.3.2 Fragmentation

The fragmentation process is required to identify each portion of the input data to be mapped into a separate output packet. Importantly, many fragmentation processes will involve very large numbers of similar pieces—for example the packets of a video or audio stream—making it highly inefficient to describe each fragment individually. In such cases, it is more desirable to describe the selection of the entire set of fragments, along with rules that determine how to divide the set into individual pieces. These rules may include a maximum fragment size or duration, a limit on the count of a particular element within a single fragment, or that particular substructures must remain unfragmented.
4.1.3.3 Packetisation

Once fragments of input content have been identified, they are inserted into packets. This process is required to identify the temporal and other parameters associated with each fragment, and also to provide any syntactical structures not present in the input data but required by the output format. Examples of such structures are XML or binary fields to structure metadata, or supplementary packet headers (e.g. aggregation mode for H.264 over RTP [58]).

4.1.3.4 Post-processing

A post-processing stage allows for custom manipulation of the output data—such as compression of XML, transcoding of media content, or encryption.

4.1.3.5 Output mapping

Finally, a mechanism is required to identify the mapping of the packets of data into the output stream, in such a way that the temporal and other characteristics of each packet are satisfied.
4.2 Implementing The Model: The Bitstream Binding Language

This section provides a detailed overview of the Bitstream Binding Language (BBL) and its use. Instead of presenting a specification that is precise but difficult to understand, a detailed example will be used to highlight the most relevant aspects of the language. The specification itself is provided in Appendix C. Additionally, while it is the language and its processing model that are the most pertinent aspects of this section, part 4.2.1 on page 112 will also discuss the architecture of the prototypical BBL server that has been developed, since this significantly aids understanding. Subsequent sections (4.2.2 to 4.2.7) discuss specific aspects of the language.

The example corresponds to the application scenario described in Section 4.1.1.1 on page 98, where a rich content container is to be delivered via multiple channels. More specifically, a broadcaster wishes to deliver a Digital TV Program defined by an MPEG-21 Digital Item (DI) that references the audiovisual content, and contains TVAnytime program metadata, usage license information, and MPEG-7 metadata. The DI is described using the Digital Item Declaration Language (DIDL), and the various metadata are to be streamed with different periods of repetition to facilitate random access [133]. Some of the audience for the TV program have a DVB-based MPEG-2 Transport Stream (TS) channel to the broadcaster, while others have an IP-based channel, and so will use RTP.

Instructions for fragmenting the DIDL and its metadata are shared between the two
channels. These instructions are declared in a BBL file (shown in Listing 4.1 on the next page) that is included by the individual files for both channels. Listing 4.1 will be discussed in more detail in subsequent sections (4.2.2 to 4.2.7).

Delivery of the video is handled by an external system for the MPEG-2 TS part of the scenario, utilising the large base of existing infrastructure in the broadcast TV domain. In such a scenario, BBL provides only the metadata, and the output mapping component essentially acts as a multiplexer, adding the metadata sections to the existing video stream.

Conversely, the BBL Processor is responsible for both media- and meta-data in the RTP scenario. Here, BBL can be thought of as a module that is added to a streaming server, to enable it to stream data in any format. As a result, additional BBL instructions are necessary to describe how the media resource is to be streamed. These instructions are shown in Listing 4.2 on page 110, and make use of a BSDL Schema (Listing 4.3 on page 111) to identify the structure of the binary media data.

While in this case delivery of MPEG-2 via RTP is already specified by a standard [25] and supported by most available streaming servers, this is not universally so. BBL provides an alternate method for content delivery that does not require a lengthy standardisation process, or server retooling, in order to support new content types. Additionally, BBL provides a very flexible way to customise content and metadata delivery for the requirements of the channel(s), user(s) and the content itself.

For other examples of delivering multimedia content with BBL, see Chapter 5.
Listing 4.1: BBL instructions for TV program fragmentation
Listing 4.2: BBL instructions for MPEG-2 video fragmentation
<xsd:schema bs2:rootElement="bitstream">
    <!−− ... −−>
    <xsd:element name="Pack" bs2:ifNext="000001BA">
        <!−− (a) −−>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="PackSC" type="xsd:int"/>
                <xsd:element name="null" type="bs1:b2" fixed="1"/>
                <xsd:element ref="mkr"/>
                <xsd:element name="SCR0" type="bs1:b3"/>
                <xsd:element ref="mkr"/>
                <xsd:element name="SCR1" type="bs1:b15"/>
                <xsd:element ref="mkr"/>
                <xsd:element name="SCR2" type="bs1:b15"/>
                <xsd:element ref="mkr"/>
                <xsd:element name="SCR3" type="bs1:b9"/>
                <xsd:element ref="mkr"/>
                <xsd:element name="ProgramMuxRate" type="bs1:b22"/>
                <xsd:element name="null" type="bs1:b7"/>
                <xsd:element name="StuffingLength" type="bs1:b3"/>
                <xsd:element name="PackStuffing" type="xsd:byte" minOccurs="0" maxOccurs="unbounded" bs2:nOccurs="mp2:StuffingLength"/>
                <xsd:element ref="SystemHeader" minOccurs="0"/>
            </xsd:sequence>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="AudioPES" type="PESType" bs2:ifNext="000001C0 000001DF"/>
                <xsd:element name="VideoPES" type="PESType" bs2:ifNext="000001E0 000001EF"/>
                <xsd:element name="OtherPacket" type="miscPacket" bs2:ifNext="000001BC 000001FF"/>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>

Listing 4.3: Part of BSDL schema for MPEG-2 system streams
4.2.1 BBL Server Architecture

On receiving a request to deliver BBL-described content, the server (Figure 4.5) parses the delivery instructions into a BBL object tree that coordinates the subsequent delivery process. BBL Objects make calls to other modules within the server in order to effect the delivery. These modules include

- Binary to BSD and BSD to binary parsers, which provide the binary abstraction function (see Section 4.2.3 on page 116);

- An XPath processor (in this case Saxon 8B). Apart from the ref attribute of `<Include>` (see Section 4.2.4 on page 117), XPath expressions are executed within the context of the current packet. This limited context allows a standard object-based XPath processor to be used efficiently;

- A streaming XPath processor for `<Include>`, to avoid the memory overhead associated with standard XPath processing on large documents. This is an
extension to Saxon which evaluates individual node-tests against elements as they are parsed, caching matching elements for later retrieval;

- An ECMAScript compiler and runtime, which provides an extension mechanism for more complex delivery requirements. A detailed discussion of the use of ECMAScript within BBL is provided in Appendix C;

- Zero or more encoder modules that provide post-processing for packets (see Section 4.2.6 on page 120); and

- Handler(s) that are responsible for placing packets on the designated delivery channel(s) (see 4.2.7 on page 120). Often, only a single channel (hence handler) is used. However, sometimes it is desirable to deliver the content on multiple channels. For example, parts of a DI could be sent on a reliable channel, and others on a channel that minimises delay.

A BBL session operates in two separate phases: set-up and execution. In the set-up phase, the main BBL instruction file is parsed, along with subordinate BBL files referenced by the main file, and any BSDL Schemas referenced by the BBL files. The parser output is the object tree, which then calls the remaining set-up operations. ECMAScript functions and XPath expressions are compiled, the various processor modules are initialised, and encoders and handlers instantiated.

When this is complete, the server enters the execution phase. Here, `<Packet>` and `<PacketStream>` tasks reserve a thread from a pool, then go to sleep until the appointed delivery time. At the correct time, the delivery instructions are executed
(see 4.2.3 to 4.2.7) and the thread either returns to the pool (for a `<Packet>`) or goes back to sleep until the next delivery time.

While execution occupies almost all of the time and resources of the BBL server, set-up and configuration are nonetheless important, and are discussed below (Section 4.2.2).

### 4.2.2 Configuration

The configuration of a BBL session is specified by the `<Register>` element and its descendants: `<Encoder>`, `<Handler>`, `<Function>`, and `<BSD>` (see Listing 4.1(a) and Listing 4.2(b)). These elements register the encoders, handlers, EC-MAScript functions and BSDL Schemata (respectively) used in the file.

Encoders and handlers may have additional descendant elements that are used to parameterise the encoder/handler instance. For example, the handler at Listing 4.2(b) has several child elements with the prefix rtp that set up the time-base and session description (SDP). The namespace and schema for these parameters are defined as part of the encoder/handler specification, as is the URI used to identify the module.

The `<BSD>` element (4.2(c)) identifies that data in a particular namespace is binary rather than XML. The attribute bsSchemaLocation locates the schema for data in this namespace, which is used to parse input binary data, and/or, generate output binary data. See below (4.2.4 on page 117) for further details.

xmlSource (4.1(a)) and binarySource attributes in the BBL instructions declare
XML or binary source documents (respectively), which provide data to a BBL session. Source documents are accessed via `<Include>` elements (and several other means, see C). Multiple sources may be declared within the document, to allow for the distributed nature of some multimedia collections, and a hierarchical mechanism is used to dereference these source documents, similar to that used by XML Namespaces [142]. For any element, the in-scope XML or binary source document is that referenced by the first `xmlSource` or `binarySource` attribute (respectively) encountered by searching the `ancestor-or-self` axis of the element [53], in order of decreasing proximity to the element (that is, first the element itself, then its parent, then its grandparent, and so on to the root node).

Parameter content and source attributes may contain literal values, or alternatively may contain an XPath expression that resolves to the desired literal value. In the latter case, the expression is delimited by “?”. For example, in Listing 4.1 on page 109, the main XML source document `TVProgram.xml` contains a reference to an external file that has the TV Anytime data. The URL for this file is contained in the specified `ref` attribute, and is used to locate the data for the `<Packet>` at (e).

Finally, a `timeScheme` attribute (4.1(d) and 4.2(a)) specifies the syntax that is used to specify temporal data within the BBL instructions. Multiple time schemes may be used in a document, and the scheme in force at any given element is determined in the same manner as the in-scope source documents. Various time schemes are provided to simplify interoperability between BBL and commonly used metadata time formats—such as the MPEG-7 time scheme, NPT, and the time codes defined
by SMPTE [73]. Time values in binary formats typically signal time using an integer field that increments with a defined frequency. The fractional time scheme is provided to cater for this case, where the frequency is specified by the integer in parentheses.

### 4.2.3 Binary Abstraction

The Bitstream Syntax Description Language (BSDL) [129] acts as an abstraction layer to allow binary resources to be handled in BBL in the same way as XML. A BSDL Schema (which is an XML Schema with additional information) describes all resources of a particular format. For example, the BSDL Schema from which Listing 4.3 on page 111 is taken describes any MPEG-2 Systems bitstream [143], such as the Program Streams typically stored in files with the extension “.mpg”.

The Pack structure (Listing 4.3(a)) is the atomic temporal unit of an MPEG-2 Program Stream, and a Pack’s time stamp is given by the System Clock Reference (SCR) field. To avoid start code emulation this field is split into parts, so it must be reassembled to read the field value. This is done in the variable $\textit{time}$ in the BBL instructions (Listing 4.2 on page 110).

While BSDL is an extension of XML Schema, this does not mean that binary resources must be transformed into XML in order to be delivered. A BSDL Schema is simultaneously a binary schema and an XML schema, and as a result it can be used to refer to binary data \textit{as if it were XML}. For further discussion see Sections 4.3.2 and A.6.


4.2.4 Fragmentation

Fragmentation is implemented within BBL by the `<Include>` element. The include element selects one or more nodes from a source document via an XPath expression (on the ref attribute). These node(s), along with a number of levels of descendants (specified by the value of depth), are included in the output in place of the `<Include>` element.

If the include element is part of a `PacketStream` declaration (see 4.2.5.2 on page 119), then the included content can be further fragmented with the rules in Table 4.1 on the next page. In Listing 4.1(c), the include element retrieves the root DIDL node, and its descendants to any depth (-1). This content is then fragmented by `FragmentAt`, so that the usage license details are repeated within the stream more frequently than other metadata, and so that metadata relevant to specific points in the program (MPEG-7 Segments) are not delivered until that time. In Listing 4.2(d), `<Include>` is used to select all of the Packs in the program stream, which are subsequently split into a single Pack per output packet.
Table 4.1: BBL Fragmentation Rules

<Size> — Declares that content should be fragmented so that its size in bytes (when serialised and/or encoded) is no greater than the given value.

<Duration> — Specifies that content should be fragmented so that the total duration of elements in any one packet is no greater than the given value.

<Count> — Specifies that content should be fragmented so that the number of nodes matched by the match expression is no greater than the given value.

<Constraint> — Applies a constraint to the fragmentation of content elements which are matched by an XPath expression. Possible constraints are

- Matched nodes must be the first content in any packet;
- Matched nodes must be the last content in any packet;
- The subtree of matched nodes must be unbroken (not fragmented into separate packets).

<FragmentAt> — Declares that a new fragment is to be created at any node matched by an XPath expression. The specific node(s) matched are present in both parent and child fragments (optionally with fragmentRef and fragmentID attributes to assist reconstruction). Any included descendants of the matched node(s) are present in the child fragment.

4.2.5 Packetisation

There are two mechanisms to define packetisation within BBL. The first—the <Packet> element—specifies how single packets are assembled. Its use is described in Section 4.2.5.1, below. Secondly, a sequence of similar or related packets may be defined more efficiently using the <PacketStream> element (Section 4.2.5.2).

4.2.5.1 Single Packet content

Single Packets are declared using the following:

- Elements from namespaces other than BBL’s (including binary namespaces)
may be directly instantiated as descendants of the Content element. This includes elements from BSD namespaces. This is the case for example with `<did:Descriptor>` and `<did:Statement>` at (c);

- Elements which are instantiated in other XML documents may be retrieved by an `<Include>` element and included anywhere in the content (c);

- Binary data may be retrieved with an `<Include>` with an XPath reference (d) according to the XML abstraction presented by the BS Schema (e); or

- The result of variable computation may be included in packet content via `<Value-of>` (not shown).

### 4.2.5.2 Packet definition as a stream

This involves specifying the entire set of content to be delivered, along with one or more rules to govern the fragmentation of the content. Content is specified using the same mechanisms as single packets (above), while fragmentation rules are declared using the elements in Table 1. Listing 4.1(d) shows an example of the FragmentAt rule, and Listing 4.2(f) of Count.

The delivery time of each packet within the stream must also be identified. This information is typically embedded within packet data or elsewhere in the source content, and is variously specified as an absolute timestamp, a temporal offset between adjacent packets, or in some cases indirectly via the hierarchical structure of the bitstream\(^2\). Each of these scenarios is accommodated by `<PacketStream>`, using

\(^2\)This is for example the case with H.264/AVC bitstreams, which contain no internal temporal
either `<DeliveryTimes>` or `<Durations>`. Listing 4.2(e) for example uses the former to specify the time extracted from the SCR.

### 4.2.6 Post Processing

Post-processing is specified with the `<Encode>` element. This references an encoder registered in the same BBL file or one of the files importing it, and can be a child of a Packet(Stream) or any element in the packet content. In Listing 4.1(g), the TVAnytime metadata is compressed using BiM [62].

### 4.2.7 Output Mapping

Each Packet(Stream) is output to the first registered handler by default (this is the case at Listing 4.2b). Alternatively, a Packet(Stream) may reference a different handler via its id, using a handler attribute, as shown in Listing 4.1(b). In the example, the handler with `id="meta"` is not declared in the file shown; it is declared in the importing file, so that the handler type can change between the scenarios (MPEG-2TS vs. RTP).

### 4.3 Evaluating The Bitstream Binding Language

Using the examples discussed in the preceding section, The following presents a per-
performance evaluation of the prototype BBL Server (Table 4.2). The server was implemented in Java and tested on a Pentium M 1.85GHz PC, with 1GB of RAM, running Windows XP and the Sun 1.6.0 JVM. Each test was performed 10 times, and the results averaged. Note that this evaluation does not consider network performance, since bytes on-the-wire are, in this case\(^3\), identical to that which would be output from an existing streaming server. Instead, this section compares the performance of the generic BBL approach with what would be expected of a format-specific traditional streaming server. All of this happens on the server, not on the wire.

In the case of the MPEG-2 Transport Stream, the BBL server manages only the metadata; the multimedia content is processed by a format-specific multiplexer. In contrast, BBL is responsible for processing the entire RTP session, requiring significantly more CPU time to do so. This highlights the trade-off inherent in providing a generic solution that can adapt to a wide range of content and metadata formats, versus the approach typical of current streaming servers, where modules are purpose-built and optimised for each format. Being a generic tool, the BBL server will not scale to the hundreds or thousands of simultaneous sessions typically observed with a conventional multimedia streaming server, such as [144]. While the implement-

\(^3\)It is certainly possible to use BBL to define new on-the-wire delivery formats, and this is one of the central uses for the language. However, the network performance of such a format is determined not by BBL, but by the design of the format itself.
tation may benefit significantly from optimisation, software written specifically for individual formats will, in general, always outperform generic software.

In a broadcast scenario, such as is typically the case with an MPEG-2 Transport Stream, this is not a critical issue since a single stream is delivered to a large audience. However, for RTP and other delivery channels, streaming infrastructure is required to handle a large number of simultaneous sessions, and hence for BBL to be viable, a mechanism is needed to allow it to scale. Two solutions to this issue have been identified:

### 4.3.1 Pre-processing of content

Preprocessing may be performed on the content, whereby the BBL process is conducted offline, and the process output placed in a file format which is easily streamed by existing servers. In this way, the format-independent BBL processing may be performed by an offline application, on a separate device. Because this offline process has no need to handle multiple simultaneous sessions, scalability is no longer a critical issue. Online processing is performed by existing streaming servers using very scalable hint-tracks.

This approach is described in Chapter 5, where the BBL processor uses a handler that is able to create hinted ISO or QuickTime files. These are essentially generic multimedia containers with hint-tracks, and are supported by most multimedia streaming servers to facilitate highly scalable multi-user streaming.
Hints are essentially byte-copy instructions, and so are processed very efficiently by the streaming server, removing the requirement that the server itself possess detailed knowledge about individual formats. The application of hint tracks to the BBL-based streaming architecture provides a complete format-independent, scalable streaming architecture. This is achieved by replacing individual format-specific modules within the hinter with a BBL processor, which is then driven by BBL instructions for hinting the multimedia content.

### 4.3.2 Pre-compilation of BBL Instructions

This is another alternative for improving the scalability of BBL processing. This option acknowledges that a single BBL description may apply to many pieces of content, and so it may be desirable to compile the BBL instructions into an efficient intermediate format (analogous to Java byte-code) ahead of time.

For example, one of the computationally expensive operations within the BBL processor is the use of BSDL as a binary abstraction. Rather than explicitly generating the XML representation of binary data, against which XPath expressions are evaluated, it is possible to compile such expressions into an algorithm that operates directly on the binary data, bypassing the XML representation entirely. This task is future work.
4.4 Conclusion

This chapter has proposed a Bitstream Binding Language (BBL), which provides a flexible and format-independent mechanism for delivery of rich multimedia Content. The central contribution of BBL is a framework for the description of mappings between a virtual container and a delivery channel, which may be a streaming format such as RTP, or a static format (an ISO/QuickTime file, for example). BBL has been adopted by MPEG as part of the MPEG-21 Multimedia Framework, and can be used to describe such mappings for any multimedia content format, enabling the delivery of format-agnostic virtual containers, and facilitating interoperability with new media types as they are developed.

A BBL Processor has been developed, and its use demonstrated for MPEG-21 Digital Item delivery over MPEG-2 Transport Streams and RTP.

In addition to providing a mapping for content into one or more output formats, BBL may be used to customise the presentation of content according to the requirements of the user, terminal and/or delivery channel. For example, BBL could be used to insert Supplemental Enhancement Information [58] into an H.264/AVC stream, for those channels/terminals which can make use of the extra information. Further, with a scalable bitstream, BBL could be configured so that it delivers different layers to different clients, depending on the parameters of each.

In the light of Reconfigurable Media Coding [116], the flexibility provided by BBL becomes even more significant. In effect, this represents a paradigm shift away from
highly rigid bitstream formats, to an approach that allows the bitstream to be completely restructured according to the requirements of the content, channel or terminal. Reconfigurable Media Coding provides a decoder that allows new video formats to be developed without extensive standardisation and application development efforts. BBL brings similar ease of extensibility to a multimedia streaming and delivery architecture.

In Chapter 5, the application scenarios outlined in Section 4.1.1 on page 98 will be implemented using the technology proposed in this chapter. That is:

- A BBL-based Format-independent streaming server;
- Using BBL to achieve generic multimedia syntax translation; and
- Specification of virtual multimedia container assembly using BBL.
Chapter 5

Erecting the tower

Applications for format-independent streaming and delivery

CHAPTER 4 set out a framework for format-independent multimedia streaming and delivery (BBL), and validated BBL against the multi-channel delivery scenario. In Section 4.1.1 on page 98 it also detailed a number of application scenarios that BBL is intended to fulfil, and this chapter demonstrates how each of these applications are implemented, including detailed examples. The first of these scenarios is a format-independent streaming server and hinter (Section 5.1). Section 5.2 on page 136 details how BBL can be used to implement generic syntax translation, and Section 5.3 on page 144 shows BBL as a means to package content into virtual containers.

5.1 A format-independent streaming server

In contrast to the streaming server architectures described in Chapter 2 (Section 2.3), a streaming server based on BBL (Figure 5.1) does not require any format-specific
software. All of the information required to stream content of a particular format is stored in a BBL description file. Whereas a hint track or extended gBSD (confer Section 2.3) describe one piece of content, a BBL description relates to all content of that format. This section describes in detail how BBL can be used as a format-independent streaming server, expanding on the example shown in Section 4.2.

This architecture means that support for new encodings as they are developed may be provided by merely disseminating a BBL description. No additional software modules need to be written, which considerably simplifies the process of providing streaming support for new formats.

The streaming server may use the BBL description to process content on-the-fly. This is useful in a live streaming situation—where the content is not available for offline hinting, or where dynamic network conditions can guide the streaming process. Alternatively, a BBL description may be used to control a hinter, processing the content offline and providing the scalability benefits of hinted streaming.

The BBL language addresses the shortcomings highlighted in Chapter 2 (Section 2.3 on page 30). It allows the identification of syntactical content structures at any level—not just Access Units—and it provides the ability to add custom headers or other data to packets as required.
5.1.1 A model for format-independent streaming

Figure 5.2 on the next page depicts the model used by BBL to enable format-independent multimedia streaming. Given an input bitstream, BBL describes how to identify the content to be included in each packet. It provides instructions to determine the timing of the packet, and the value of header fields. The latter may involve both standard headers (such as the RTP header), and payload-specific headers, where it is necessary to define both the syntax and the values of each field.

Identification of packet content: In general, multimedia bitstream formats are made up of numerous layers of syntactical structures. In a streamed delivery scenario, the packetisation of the bitstream must proceed on the basis of these structures, in order to ensure timely delivery and facilitate error resilience [58, 59]. A format-independent mechanism is therefore required that is able to identify the syntactical elements of a bitstream, such as XFlavor [30] or BSDL [28]. The examples below use BSDL for this purpose, but it is also possible to use XFlavor or other such languages.
To specify packet content, an XPath expression selects the set of content to be packetised, and a number of rules are applied to determine how to divide the set into individual packets. The available rules are based on the requirements of numerous use cases, including IETF specifications for various content formats [58, 59]. They may include a maximum packet size or duration, a limit on the count of a particular structure within a single packet, or that particular sub-structures must remain whole.

**Timing information:** Some content formats have a constant or variable packet duration which may be read or inferred from the bitstream (for example, Theora, MP3, MPEG-4 Visual). Others use explicit timestamps (such as MPEG-2 Program Streams). H.264/AVC, on the other hand, contains no internal temporal information; this must be provided externally.

BBL supports all of these cases. Packets are placed on a timeline beginning at $t_0$ where the delivery time $t$ of packet $n$ is given by

$$t_n = t_{n-1} + \Delta_{n-1}$$ \ldots (5.1)
where $\Delta_{n-1}$ represents the duration of a packet (Figure 5.2 on the facing page). Both $t_n$ and $\Delta_n$ may be explicitly specified within the BBL description. Typically, only one is used for a particular instance, however there are some situations where resynchronisation points in the bitstream may have an explicit timestamp, while other packets are given a duration offset.

Temporal information is declared in BBL using two XPath expressions. The first identifies the bitstream segment(s) to which the temporal parameter is to be applied. The second describes how the timestamp or duration is calculated from the fields within the bitstream segment (which have been identified by BSDL), and/or values which have been stored from other sections of the bitstream.

**Standard Header data:** On the Internet, RTP is used almost exclusively as the streaming protocol. However, BBL was designed for use in multiple domains (such as Digital TV, where MPEG-2 Transport Streams are typically used), and provides a mechanism to specify alternative output stream handlers. This handler mechanism is also extensible, so that new streaming protocols may be easily integrated into the BBL framework.

A handler receives the data for each packet, along with its delivery timestamp, and other parameters defined specifically for the handler. For the RTP handler, this includes the timebase, payload type, SDP (Session Description Protocol) [145] data, and marker bit. These parameters provide the values for some of the RTP header fields. Other RTP header fields, such as the sequence number and SSRC (Synchronisation Source identifier) [131], are not media specific—they are set by the streaming
server without information about the content.

**Payload-Specific headers:** The mechanism used to specify packet content may contain multiple separate elements. This allows payload-specific headers to be added to packet data. BSDL is used to specify the structure of the header, and XPath expressions to calculate the field values.

### 5.1.2 BBL For H.264/AVC over RTP

H.264/AVC [12] is a recent video encoding format used as an example application for BBL, since it has a number of characteristics distinct from previous coding formats which make generic streaming more challenging. These include parameter sets and a lack of internal timing information.

A H.264 stream is made up of sequences of Network Abstraction Layer (NAL) Units containing slices of picture data, interleaved with parameter set NAL units and other supplementary data. In general, each NAL unit in the input bitstream is carried in a separate RTP packet [58]. In the BBL description (Listing 5.2 on the next page), this is accomplished by selecting all of the NAL units to be packetised using the include element, then applying fragmentation rules to separate the NAL units into individual packets (not shown).
<register>
  <handler id="default" type="RTP">
    <rtp:sdp>sdp.txt</rtp:sdp>
  </handler>
</register>

<packetStream>
  <params>
  </params>
  <contentTemplate>
    <include ref="/avc:h264/avc:slice | /avc:h264/avc:parameterSet"
            depth="−1">
      <timing>
        <delTimes match="." value="$delTime"/>
      </timing>
      <fragmentation>
        <count match="." value="1"/>
      </fragmentation>
    </include>
  </contentTemplate>
  <variables>
    <!--
    <assign name="delTime"
            value="if ($newAU)
                      then $delTime + $framePeriod
                      else $delTime"/>
    
    <assign name="expectedPicOrder"
            value="if ($nalType = 5)
                      then 0
                      else if ($newAU) then $expectedPicOrder + 2
                      else $expectedPicOrder"/>
    
    <assign name="timestampOffset"
            value="if (.avc:h264/avc:slice)
                      then $frameTime ∗ ($picOrder − $expectedPicOrder) div 2
                      else $timestampOffset"/>
    
    </variables>
  </packetStream>

Listing 5.2: BBL Instructions for streaming H.264/AVC content
In order to derive timing information for each NAL unit, their association to Access Units (AU) must be identified. The bitstream may contain a specific AU delimiter which simplifies this process but its presence is not guaranteed and so cannot be assumed. Consequently, the general process specified in clause 7.4.1.2.4 of [12] is used to calculate the Boolean variable $newAU$, by detecting changes in certain field values between one slice NAL unit and the next (these fields include $\text{nal\_unit\_type}$, $\text{frame\_num}$ and $\text{pic\_parameter\_set\_id}$).

The NAL Units in an AU have the same delivery time $\text{delTime}$—based on an external frame rate. The RTP header timestamp, however, must be offset from the delivery time according to the display order of the pictures. This is implemented in BBL by comparing the $\text{pic\_order\_cnt}$ field ($\text{picOrder}$) to its expected value ($\text{expectedPicOrder}$).

### 5.1.3 Results

Functional testing validates the correct operation of our format-independent system. This testing was carried out by executing the BBL instructions in Listing 5.2 on the preceding page against H.264/AVC sequences in each of the format’s profiles. The tests were conducted using a QCIF input sequence of 382 frames (15.3 seconds at 25fps). The configurations used were:

(a) Baseline profile with NAL size limited to 100 bytes (a profile targeted towards mobile applications [146]);
Table 5.1: CPU and Memory utilisation for prototype BBL processor with H.264/AVC over RTP

<table>
<thead>
<tr>
<th>Test configuration</th>
<th>NAL unit count</th>
<th>% CPU utilization</th>
<th>Memory usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Baseline</td>
<td>4555</td>
<td>23.0%</td>
<td>1.62</td>
</tr>
<tr>
<td>(b) Main</td>
<td>459</td>
<td>3.1%</td>
<td>0.94</td>
</tr>
<tr>
<td>(c) Extended</td>
<td>1146</td>
<td>5.5%</td>
<td>1.16</td>
</tr>
</tbody>
</table>

(b) Main profile, (introducing bi-predicted frames), NAL size 1500 bytes; and

(c) Extended profile using data partitioning—an error resilience feature provided by H.264 where each slice is split into 3 portions with varying loss importance.

In all cases, the BBL server produced a bit-exact replica of the RTPdump output of the H.264/AVC reference software [147], in faster-than-real-time (Table 5.1 and Figure 5.3 on the next page). The Baseline profile video had more NAL units, and therefore more packets, by an order of magnitude (compared to the main profile clip). Unsurprisingly, this resulted in substantially greater time complexity (CPU utilisation) than the other clips. Space complexity (memory usage), however, grew only slightly. To improve scalability in an application requiring small NALUs, AU delimiters could be employed to reduce time complexity, or offline processing (hinting) used with greater priority. These results show that the prototype BBL server can feasibly be used to deliver a broad spectrum of H.264/AVC content over RTP as-is in a multi-session server. It is expected that further optimisation will continue to improve the scalability of the system.

---

1. The prototype BBL processor was implemented in Java and tested on a P4 3.0GHz PC, 1Gb of RAM, Windows XP & Sun 1.5.0_04 JVM. The reported memory usage excludes that used by the JVM.
Figure 5.3: CPU and Memory utilisation for prototype BBL processor with H.264/AVC over RTP

5.2 Generic multimedia syntax translation
(cf. Section 4.1.1.2 on page 99)

Aside from the multitude of codecs and container types used for the content itself, an increasing number of formats are being defined for multimedia metadata. Many of these have binary syntax, such as ID3 [76], EXIF [138], and Matroska [148]. Others use XML to structure the metadata—MPEG-7 [136], TVAnytime [130], and so on. The human-readable nature of XML also means that many groups define private XML grammars for communicating multimedia metadata.

BBL may be applied to the more general problem of how to enable existing devices to consume multimedia content published in new formats. As an example (Figure 5.4 itself.
on the next page) a BBL description can convert a MAF file (Section 2.3.3.2 on page 41) into either a) an MP3 file with ID3 (version 1 or 2) metadata, b) streamed content and metadata for delivery to a 3GPP device, or c) a proprietary format for some other device. This makes it feasible for content providers to deploy the new MAF format and still allow the content to be consumed on legacy equipment. The software which performs the translation—the BBL processor—is generic with regard to multimedia formats, and all that is required to support a new output format (whether file-based or streamed) is an additional BBL description. BBL does not directly describe the transcoding of media, but instead provides a mechanism to plug in post-processing modules which can be used for this purpose.
5.2.1 An example generic syntax translation

This Section demonstrates how BBL may be applied to one of the scenarios from Figure 5.4—conversion of a Music Player MAF file to ID3v2 [76] and MP3 [1]. This operation is shown in further detail in Figure 5.5. BBL processor inputs are

(a) the MAF file;

(b) a BBL instruction file; and

(c) BS Schemata to expose the structure of the input and output data.

5.2.1.1 MAF Bitstream Syntax Schema

Listing 5.3 shows a excerpt from the BS Schema that describes the structure of a MAF file (as depicted previously in Figure 2.12 on page 42). Each box in a MAF file
Erecting the tower

Listing 5.3: MAF BS Schema excerpt

is an extension of either maf:BoxType or maf:FullBoxType (which itself extends maf:BoxType).

Some XML Schema datatypes—unsignedByte for example—have a direct corre-
spondence to a binary representation (8 bits in this case). Other datatypes may be
derived from these base types—such as maf:hex3—via the XML Schema facets
(length, maxExclusive, and so on) and also via a number of BSDL pseudofacets
including bs2:length. The facet bs2:length allows the size of a datatype to be de-
determined dynamically by the result of an XPath expression.

The complex type definition for the metadata box is also shown (maf:MetadataType),
to demonstrate some of the mechanisms used to specify bitstream structure. This box
extends maf:FullBoxType, so the first items in the box are a size field, four-character
id code, version number and set of flags. Following this, a metadata box may contain
a number of sub-boxes, in any order. This is modeled in the schema by the choice par-
ticle, where the choice between the possible elements is determined by the value of
the sub-box four character code (specified by bs0:lookAhead and bs2:ifNextStr—).
The overall size of the choice particle is specified by the XPath expression within
bs0:layerLength, which references the box size field and subtracts the size of the
box header.

5.2.1.2 MP3/ID3 Bitstream Syntax Schema

Part of the BS Schema for MP3 and ID3v2 is shown in Listing 5.4 on the facing page.
It is used to specify how the structure of ID3 metadata within an MP3 file, which is
the output of the BBL process in this example (Figure 5.5). Numerous features of
the BS Schema are similar to those described above (Section 5.2.1.1 on page 138),
and need not be described again in detail.
Erecting the tower

Listing 5.4: MP3/ID3 v2 BSSchema excerpt
The macro-structure of the file is an optional ID3 tag (detected by the presence of the string “ID3”), followed by the MP3 data. Because the MP3 data is to be written to a file, its internal structure need not be specified explicitly, and so it is represented by a `bs1:byteRange` datatype, which provides an offset and length for the data within the referenced file (this file is specified by the `bs1:bitstreamURI` attribute).

The size of the ID3 tag payload is stored as a “synchsafe integer” [76], which is essentially four 7-bit fields interspersed with single zero bits. Consequently, computation of this value is more involved than was previously the case (the `bs2:layerLength` attribute on the sequence particle). The ID3 payload may contain numerous frames; only `TextFrameType` is shown. This consists of a one-byte encoding indicator, and a null-terminated string which may be empty.

### 5.2.1.3 BBL Instructions

Listing 5.5 shows part of the BBL instructions for translating MAF files into a legacy format (MP3 with ID3v2 metadata) so that the new format may be consumed by existing devices. The output of the processor is specified by the content element (a), which is converted to its binary representation (specified by the MP3/ID3 v2 BS Schema—see Section 5.2.1.2 on page 140), and post-processed by a `deformatMP3` operation.

The majority of the BBL instructions (of which only a small subset are shown in Listing 5.5; the complete instructions are listed in Appendix F) perform the task of extracting the metadata from the MPEG-7 description within the MAF file, and insert-
Listing 5.5: BBL Instructions for MAF to MP3/ID3

ing it into the appropriate ID3 frames. This is shown for the ID3 text frame with the ID TIT2 (b), which corresponds to the MPEG-7 songTitle descriptor (&MPEG7PATH; is an XML entity declared elsewhere in the document). The size field of the text frame is populated via the XPath variable $id3SongTitleLength,
which is defined in the variables section of the packet definition (d). ID3 text frames for other MPEG-7 descriptors are similarly specified, and the image data is inserted into an ID3 APIC (Associated Picture) frame, using a similar mechanism to that below.

The id3:mp3Data element contains the audio data. The location of this data within the MAF file is retrieved using the references in the DIDL and iiinf/iloc boxes (c). This location (an offset and length) is inserted the element content, which was specified as a bs1:byteRange type (in the BS Schema for ID3—Listing 5.4). This datatype extracts the byte array from the input file and copies it to the output.

5.3 Virtual container assembly
(cf. Section 4.1.1.3 on page 101)

This section describes extensions to BBL to enhance the scalability and flexibility of the framework, by providing support for outputting the widely-used ISO [56] and Quicktime (QT) [5] file formats. These are essentially generic multimedia containers, and provide hint-tracks which are used by most multimedia streaming servers to facilitate highly scalable multi-user streaming (Figure 2.6 on page 33). The hint track allows much of the processing involved in streaming content to be performed offline—by the hinter—to generate hints, which are essentially byte-copy instructions. The hints may be processed very efficiently by the streaming server, and they alleviate any requirement that the server itself possess detailed knowledge about in-
dividual formats. The application of hint tracks to the BBL-based streaming architecture (Figure 5.1 on page 129) provides a complete format-independent, scalable streaming architecture, by replacing the individual software modules within the hinter with a BBL processor, which is driven by a data file containing BBL instructions for hinting each particular format. Furthermore, the same mechanism allows BBL to be used to describe how content can be packaged into virtual containers, as described in Section 4.1.1.3 on page 101.

5.3.1 ISO/Quicktime File Output Handler

To implement the format-independent, scalable streaming (Section 4.3.1 on page 122) and virtual container assembly (4.1.1.3 on page 101) application scenarios, the BBL processor must be able to output valid ISO format files, including sample tables and hint tracks. As discussed, the BBL architecture is designed to be extensible, via its output handler mechanism, so ISO file output may be provided by specifying an appropriate handler. The following provides an overview of the ISO file handler design. A full specification may be found in Appendix H, and Section 5.3.2 (below) presents an example which may aid understanding of the discussion.

In designing the handler, a number of requirements must be observed: the handler must be format-independent—that is, it is able to receive input content of any form and place it within an ISO file. Secondly, the handler must be extensible; it must allow arbitrary ISO file structures to be defined as required, enabling the BBL processor to create ISO files which conform to file formats which may not yet exist (such
as the various MAFs). Finally, the handler must be as simple as possible—the BBL author must not be required to describe details which may otherwise be computed automatically (such as the data in sample tables and hint tracks).

ISO files have an object-based structure, via a set of nested “boxes” (Figure 5.6). Each box has a particular semantic, and contains data fields and/or other boxes. An ISO file has a mandatory file box (ftyp), which specifies the type of content within the file, and usually a movie box (moov), which contains a large number of boxes for the low-level metadata necessary to understand the raw content. This metadata includes sample tables, timing data, and hint information. Conversely, a metadata box (meta) contains high-level metadata such as an MPEG-7 description of the content, or links to related content. The content itself is stored in the media data box (mdat).

The handler is able to make some assumptions about the box structure of the file it is to create—for example, certain sub-boxes are mandatory or contingent upon various conditions, including many within the movie box. Other aspects of the file structure must be explicitly specified within the BBL description, such as the number and type of tracks within the file, the movie timescale and sample descriptors. These
Figure 5.7: Static and global characteristics are specified by a global parameter set. Parameters specific to a packet of media data are specified with that packet attributes are specified as part of the global parameter set provided when the handler is initialised (Figure 5.7).

The global parameter set establishes the structure of the output file, provides values for header fields, and initialises sample descriptors. It also provides IDs for each track, sample descriptor and/or metadata item to allow packet-level parameters to identify where the packet belongs. Figure 5.8 on the next page shows the schema of this parameter set using XMLSpy-style notation [149]: each box is an XML element, and most elements contain attributes which provide values for various fields in the ISO box structure (attributes are not shown due to space limitations). The complete XML Schema for the global parameters may be found in Appendix G. The structure of the parameter set is related to the hierarchical box structure of the ISO file format. However, only those details which must be parameterised by the BBL description are exposed—many parts of the file, such as sample tables and hint tracks, are created automatically by the handler, and need not be specified by the parameter set.

The handler is designed to be as generic as possible—sample descriptors may be
specified by extending the base classes with format-specific information using BSDL. Similarly, a generic box element is provided which may be used to add additional boxes using BSDL syntax. This means that the handler does not need to be modified to support new content formats.

Once the global structure of the ISO file is defined, the handler is able to process packets of content. Each packet must include parameters associating the content with a particular track, as well as sample-specific information such as a composition time offset and hint track information. The handler uses this information to make appropriate entries in the track sample table, associated hint track, and/or item location and information tables (iloc/iinf—used by certain MAFs [70], see Section 5.2 on page 136). Figure 5.9 on the next page shows the schema for the packet-level
5.3.2 Example—Scalable Generic Streaming of H.264/AVC over RTP

This section presents a detailed example of packaging a raw H.264/AVC [12] stream within an ISO file. This ISO file may then be delivered using any existing streaming server supporting the ISO format—for example, the Darwin Streaming Server [144].

Listing 5.6 presents a BBL description for this operation which shows the instructions pertinent to the ISO handler. The handler is registered at the beginning of the file, and provided with global parameters which define the structure of the output. These include declaration of (a) file-type branding, (b) the video track (other tracks may be defined in like fashion), (c) the AVC sample descriptor, (d) XPath expressions to calculate field values, and (e) the hint track.

The sample descriptor is one of several boxes within an ISO file which have a different structure depending on the input format. In order to retain format-independence, the structure of such boxes are specified using BSDL, and the value of individual
fields via XPath expressions.

The handler also receives a set of parameters with each packet. The primary function of these parameters is to associate the packet with a particular (f) track, and (g) sample descriptor, as well as provide packet-level metadata such as (g) sample boundaries and (h) hint track data. The packet content is identified by the XPath expression (i) and fragmented according to (j). This fragmentation rule specifies that there should be at most 1 root object (the slice NAL unit) in each packet. Other types of fragmentation rules are possible, and could be used to instantiate the NALU aggregation mode specified by [58]. The delivery time for each fragment is provided by the $delTime variable (k), and the calculation itself performed at (l) (not shown).

5.4 Conclusion

This chapter has demonstrated how the Bitstream Binding Language fulfils the application scenarios set out in Section 4.1.1 on page 98.

It has shown how BBL may be used to implement a format-independent streaming server (Section 5.1 on page 127). This facilitates multimedia interoperability in the face of newly developed content formats, by enabling streaming support via a data file (the BBL instructions), rather than requiring new software to be developed to support the new format. BBL can be used to process content on-the-fly, or offline to produce highly scalable hint tracks whilst still providing format-independent streaming. This approach has been tested using the H.264 video format, and produces RTP
Erecting the tower

151

streams with the correct timing and data. The prototypical BBL processor tested performs nearly two orders of magnitude faster than real-time, and optimisation would improve this further.

Section 5.2 on page 136 demonstrated how the Bitstream Binding Language may be used to enhance interoperability for rich multimedia content, by providing a generic mechanism for translating the content from one syntax to another. Using the Music Player MAF format as an example, the section shows how BBL can extract fragments of binary and XML data and insert them into a different output format, so that existing devices are able to render the content.

Finally, 5.3 on page 144 presented an extension to the BBL framework which allows it to act as a format-independent hinter and virtual container assembly mechanism. This enables a generic streaming architecture which may support new content formats as they are developed simply by disseminating a data file—a BBL description for delivering content in the new format. Such a solution obviates the need for any modification of existing streaming infrastructure in order to deploy new content formats, and consequently simplifies interoperability with these formats.

The following chapter will consider media delivery from a different perspective, utilising semantic metadata to improve decision-making about streaming scalable content.
Listing 5.6: Abbreviated BBL instructions for scalable generic streaming of H.264/AVC on RTP
Chapter 6

What are you trying to say?

Format-independent semantic-aware streaming and delivery

“[Elizabeth Bennett] looked at her father to entreat his interference, lest Mary should be singing all night. He took the hint, and, when Mary had finished her second song, said aloud, ‘That will do extremely well, child. You have delighted us long enough.’ ”

— Pride and Prejudice, 1813, Jane Austen [150]

Users automatically associate many layers of meaning with the media content they consume, yet computers have barely begun to scrape the surface of this information. For example, consider the passage above. The subtle exchange of glances between Elizabeth and her father would be readily apparent to most human observers, but it is unlikely that a computer processing a video of the scene would be able to recognise their meaning. Furthermore, while the double-entendre in Mr Bennett’s remark would be clear to most human listeners, efforts toward computational recognition of
this and other commonly used modes of speech are in their infancy [151].

Other research communities are developing means to communicate such semantic information (whether computed or manually generated) in ways that are able to transcend the original context of the information. This work—originating from Knowledge Representation, but more popularly known as the Semantic Web—has provided languages such as the Resource Description Framework (RDF) [111] and Web Ontology Language (OWL) [33] which can be used to express concepts in such a way that “this picture has many buildings” may also imply that “it is a cityscape”, and “it contains man-made objects.”

Recent multimedia coding formats developed by MPEG and ITU-T such as Scalable Video Coding (SVC) [15] and Scalable-to-Lossless Coding (SLS) [16] offer the ability to dynamically adapt their bitrate to changing conditions. Current systems perform this adaptation on the basis of static channel parameters such as terminal and network capabilities [117] or dynamic estimation of channel capacity [101]. There are, in fact, numerous examples of using content semantics to identify the best way to adapt content to dynamic conditions: Section 6.1 describes this in further detail. However, while others have proposed specific semantics to be used in the delivery process, there exists no generic system for connecting arbitrary semantics to the adaptation/delivery process. Such a system is the subject of this chapter, and it provides a link between higher (semantic) and lower (delivery-oriented) layers of the abstraction model in Figure 1.2. The combination of semantics and multimedia delivery in a generic fashion is, unsurprisingly, a complex operation that draws on a
What are you trying to say?

Figure 6.1: A framework for semantic-aware multimedia delivery

Figure 6.2: Coupling multimedia semantics and a delivery optimisation process requires numerous disparate technologies

number of disparate fields (Figure 6.2). Wherever possible, sufficient background has been provided to appreciate the concepts discussed, however, the reader is referred to the relevant citations for a more comprehensive treatment of the fundamentals.

6.1 Semantics in the delivery process

A typical news report (Figure 6.3 on page 158) provides a good example of how content semantics could be useful in multimedia delivery. News reports often have a fairly consistent structure, beginning with a studio introduction, then footage of the event (often with commentary overlaid on top of audio from the event), using subtitles if subjects are speaking in a foreign language, and sometimes concluding with
further studio footage. As a report proceeds through these various stages, the relative importance of the audio and video varies. For example, in the studio introduction, virtually all of the semantic content of the presentation is carried in the audio. On a low-bandwidth (e.g. mobile) channel, reduction of the frame-rate in this region would have little impact on the transmission of the content semantics. When the report cuts to on-site footage, a much greater proportion of the semantic content is carried by the visuals, though the amount would vary from one report to another. If subtitles are overlaid on the video, then virtually all of the semantic content is conveyed by the video, and bits spent transmitting audio in this section will contribute much less to the successful delivery of the content semantics.

As a second example, instead of comparing relative semantic importance along the modal dimension (as is the case above), similar comparisons could be made on the temporal axis. Here, temporal segments of the news report would be annotated with an indication of the relative importance of the segment to the story as a whole. This would allow users to receive a short “digest”, the full story, or something in between. A similar approach could work for coverage of sporting events.

Numerous other types of semantic metadata have been identified which can assist in delivery optimisation, as discussed in Section 2.4.4 on page 55: Bertini et al. [105], Xu et al. [106], and Baba et al. [107] all argue that applying the same adaptation operation to different parts of a multimedia presentation will have differing effects in the perceptual quality of the presentation as a whole. Bertini and Xu propose adaptation on the basis of semantic classification of sporting events into categories,
while Baba [107] proposes adaptation of speech signals on the basis of sound level. He argues that regions of (relative) silence within a speech signal carry no semantic information, and as such may be truncated during playback. Finally Cranley and Murphy [104] suggest the use of measures of the temporal and spatial complexity of a video to trade-off frame rate with resolution, to achieve a so-called Optimum Adaptation Trajectory.

The semantic-aware content delivery framework proposed in this chapter provides a way to incorporate these and other semantics into the delivery of multimedia. This is achieved in a way that is flexible enough to support the increasingly diverse range of formats, semantics, and networks that are used (or useful) for content delivery [10]. Before a detailed discussion of this framework, Section 6.2 (below) identifies a number of key features that are necessary for the framework to successfully address the challenges posed by this diversity. The proposed framework itself is then detailed in Section 6.3, along with an analysis of existing work that is able to fulfil some constituent parts. Section 6.4 describes subjective testing validating the approach, and Section 6.5 offers some concluding remarks.
6.2 Features

Multimedia semantics is an extremely diverse field. Similarly, multimedia delivery is categorised by an exponentially growing array of devices that access and process multimedia, and an increasing number of formats in which multimedia content is encoded. Given this complexity, this section argues that successfully combining semantics and delivery requires a flexible approach, where the semantics and formats used are not hard-coded, but instead described declaratively as content metadata.

6.2.1 Format-independence

The present, exponential rate of growth in both multimedia devices (hardware and software) and content formats is increasing the difficulty of maintaining interoperability. To be effective in this environment, a semantic-aware delivery framework must support content that is encoded in any current, or future, format. As has been shown (Chapter 4), for many multimedia operations, it is possible to abstract the
format-specific details of any given codec into a data file (hardware and software is then format-independent). This greatly simplifies interoperability, since a new content format can be integrated into existing devices merely by disseminating a file that describes its format-specific details. Crucially (given the exponential growth in the range and diversity of multimedia devices) no modification of hardware or software is necessary.

This argument also holds for the syntax in which semantic metadata is encoded; as discussed (Section 6.3.2.2 on page 174) there are many syntaxes used to encode the metadata needed for semantic-aware delivery. Further, as is the case for content formats, the framework must also cope with new metadata formats, as they are developed. In response to these observations, methods for adapting metadata syntax without requiring changes to software or hardware have been proposed (Chapter 5) and are important to allow a semantic-aware delivery framework to be as widely applicable as possible.

6.2.2 Semantic-independence

The range of semantics that people associate with media content is effectively infinite. The examples cited in Section 6.1 therefore represent just a small sample of the possibilities for using semantics to guide multimedia delivery. As such, it is important that a semantic-aware delivery framework not be limited to using a small, defined set of concepts.
6.2.3 Multiple optimisation algorithms

As seen in Section 2.4.3 on page 53, a considerable number of algorithms have been developed for optimising the Rate-Distortion (R-D) performance of multimedia delivery [101–104]. These algorithms vary in their guarantees of tractability, complexity, and the range of metadata required as inputs to the process. As a result, different algorithms may be preferable in certain scenarios, and so flexibility in this regard is an important characteristic of a semantic-aware delivery framework.

6.2.4 Segmentation and association

The examples cited in Section 6.1 on page 155 differentiate the semantic importance of segments of content that have been segmented along numerous axes. The most straightforward is with the sporting analysis and speech sound-level concepts, where some temporal segments are more important than others. This is also the case in the news example, but here a distinction is also made along the mode axis: in some temporal segments the video has more semantic importance, in other segments it is the audio. Cranley and Murphy [152] distinguish between semantic importance along the temporal and spatial axes. Although it has not been widely utilised, MPEG-4 [153] generalises this concept still further by introducing other modes (text, graphics, and hybrid coding), and additionally provides the ability to arrange multiple audiovisual “objects” within a scene. In such a scenario, it may be highly advantageous to attach (time-varying) semantic importance to each of these modes and objects.

Clearly, the utility of a semantic delivery framework would depend considerably on
Figure 6.4: An architecture for Multimdia Delivery that incorporates content semantics it having the flexibility to segment content along all of these (and potentially other) dimensions. After segmentation, such a framework would need to be able to associate semantic and other metadata with these segments, in such a way that they can be input to an algorithm that makes the trade-offs described.

6.3 A framework for format-independent semantic-aware multimedia delivery

Figure 6.4 depicts the proposed architecture of a semantic-aware delivery framework. As proposed by Chakareski [154], the Rate-Distortion Optimisation (RDO) process is split into two parts: generation of R-D metadata is performed offline by a *hinter*, minimising the amount of computation that must be done by the real-time
delivery node. The present work extends this concept by proposing an architecture for the hinter that is format-independent, for the reasons outlined earlier in Section 6.2. Additionally, the hinter in Figure 6.4 provides for Semantic Distortion (see below, Section 6.3.2.2 on page 174) to be combined with the “classical” approach to distortion where decoded samples are compared to the samples that were originally encoded, using a measure such as (peak-)SNR, referred to as Sample Distortion.

### 6.3.1 Delivery Node

With the static content analysis performed offline by a hinter, a delivery node (Figure 6.5) is left only to decide whether and when to forward, drop or truncate (where applicable) each packet. That decision is made on the basis of some type of rate-distortion optimisation algorithm, which takes as its inputs feedback about the channel condition, and metadata from the semantic hinter. These elements are described in the following sections.
6.3.1.1 R-D Optimisation

There are a number of rate-distortion optimisation algorithms. Different algorithms perform better in particular scenarios, and so this semantic-aware framework avoids prescribing one over another. Instead, the framework allows the most suitable algorithm(s) to be implemented on any given delivery node. The most relevant algorithms have been discussed previously in Section 2.4.3; this section briefly summarises that discussion in the context of the Delivery Node.

Chou [101] proposes the use of classical optimisation of R-D performance $D(R)$ by minimising the Lagrangian $D + \lambda R$ for some $\lambda$. The formulation of distortion must consider the error probability-cost functions for each unit of data, as well as the interdependencies between Data Units, since descendent packets (e.g. any motion-compensated frame, or enhancement layers in SVC) generally cannot be decoded if their ancestors are not received.

Chakareski et al. [102] note that although the algorithm proposed by Chou is theoretically optimal and suitable for certain applications, it comes at the cost of significant computational complexity. Consequently, Chakareski proposes a low-complexity approximation of the lagrangian optimisation problem, by ignoring interdependencies between Data Units and instead considering the distortion imposed by packet loss on subsequent packet distortions to be additive.

Eichhorn [103] suggests the opposite approximation: Chakareski ignores actual dependencies; Eichhorn ignores actual distortions, and asserts that dependency alone
may be sufficient. Finally, Cranley and Murphy [104] trade temporal resolution against spatial resolution and use subjective testing to arrive at a so-called Optimum Adaptation Trajectory.

### 6.3.1.2 Serialisation of hinter metadata

On the one hand, a binary syntax could be specified for hinter metadata, in order to maximise space efficiency over-the-wire\(^1\). However, this makes extension of the data set (as is likely inevitable as new optimisation techniques are developed) difficult to achieve without breaking existing implementations. For this and other reasons, most recent metadata uses XML rather than binary syntax, because of the ease with which it is processed and parsed, despite its inherent verbosity. As it turns out, it is possible to achieve most of the benefits of both, using the so-called Binary format for Metadata (BiM) [62]. BiM uses XML Schema to provide efficient binary encodings of XML data. This means that the R-D metadata can be created and processed as XML, but if it must be transmitted, BiM can achieve transmission efficiencies close to those of a dedicated binary syntax. Furthermore, at the downstream node, the binary representation may be parsed directly, without decompression, avoiding any additional time complexity.

\(^1\)That is, when this metadata must be transmitted on-the-wire, which is only the case if the Delivery Node is remote from the content, for example if it is a gateway node (Figure 1.1)
6.3.1.3 Summary

In deciding whether to forward or drop each packet as it is received, the Delivery Node utilises some sort of optimisation algorithm. Several of these were discussed above (Section 6.3.1.1). Depending on the algorithm chosen, different metadata is required from the R-D Hinter, although a common subset of data including $\Delta_{time}$, $\Delta_{rate}$ and segmentation is required for the forward/drop routine. Otherwise, this metadata may contain the set of distortion increments $\Delta D_l$, the Data Unit dependence graph, or SI and TI values$^2$. This also points to the need for a negotiation process between the Delivery Node and the node holding the content to identify the desired optimisation algorithm based on the available metadata, although in some cases a node may be able to generate missing metadata on-the-fly (with the concordant time penalty).

If the Delivery Node is remote from the content and metadata, for example if it is a gateway node, then it may be desirable to minimise the bandwidth used by the R-D metadata, by utilising BiM [62] to binarise the data. In this case, the Delivery Node would use a BiM parser that directly interprets the binarised metadata and passes the output data points directly to the RDO algorithm.

6.3.2 Semantic R-D hinter

The role of the hinter (Figure 6.6 on the next page) is to prepare the metadata needed by the R-D Optimisation algorithm. This metadata can then be stored in a file (such

$^2$see Section 2.4.3 on page 53 for a detailed discussion of these.
Figure 6.6: The semantic hinter computes R-D metadata based on content syntax and semantics.

as an ISO [56] or Quicktime [5] container) for later use, or transmitted with the content to a local or remote delivery node. The hinter itself is composed by elements that analyse the semantics and the syntax of the content. The former (semantic analysis) obtains the higher-level characteristics of the content which are typically not evident in the compressed domain, but must be identified from the original (reference) samples or entered manually. Section 6.3.2.2 on page 174 considers semantic analysis in greater detail. On the other hand, syntactic analysis (discussed in Section 6.3.2.1 on the next page) extracts the interdependency, temporal and scalability metadata that are direct parameters of the compressed bitstream.
6.3.2.1 Syntactic Analysis

Syntactic analysis is the process by which the hinter exposes the syntactical elements of multimedia that are needed by a given RDO algorithm. This occurs in two stages. First, the underlying syntactic structure of the content must be exposed so as to provide access to the internal data fields. In this work, binary schemata (Chapter 3) are used to achieve this functionality. Secondly, a mapping must be made from the arbitrary raw data fields exposed by a schema, to the specific concepts needed for RDO.

Binary Schemata

Recent coding formats utilise increasingly complex multi-layer structures to encode media in ever-fewer bits. As a result, identifying the timestamp, interdependencies or even byte-boundaries of an encoded Data Unit requires significant parsing. In most systems, this parsing is performed by format-specific software or hardware, that is, the format of the codec is “hard coded” into the parser. However, because the number of coding formats is large and growing, such a hard-coded approach makes it increasingly difficult to maintain interoperability with the available coded content.

An alternative is to use a reconfigurable or generic parser for syntactic analysis, where the specific syntax of individual codecs is stored in a schema data file. Support for additional formats may then be added via a new file, rather than new hardware or software (see Chapter 3).
Figure 6.7: A binary schema can be used to expose the structure and data in a binary bitstream as OWL/RDF.

Lehti [155] shows that the object-based structure of XML Schemata (and the XML data they describe) means that it is possible to map from XML Schema complex and simple types (which directly or indirectly represent binary structures in the case of a BSDL Schema) to OWL classes and properties (respectively).

This approach is far from elegant, because XML Schemata describe syntax whereas OWL [33] and RDF [111] describe semantics, and mixing the two in this way can lead to significant ambiguity. Nonetheless, it is useful, since combining it with one of the binary schemata languages described above allows binary data to be directly integrated with OWL/RDF-based data). This means that binary content may be processed and queried as if it were RDF triples. Figure 6.7 depicts an example of the
What are you trying to say?

approach. A BSDL schema describes the binary structure of an SLS bitstream (the example shows an SLSSpecificConfig [120] config structure). At the same time, this BSDL schema describes the structure of an XML representation of the syntax of the binary bitstream. Lehti’s technique is then used to map the BSDL/XML schema into OWL classes, and to map the XML metadata into RDF triples. This allows the binary structure of the SLS bitstream, along with the data it contains, to be reasoned on and combined with other OWL/RDF semantic data about the content. ³

Mapping Rules

The metadata exposed by using a binary schema will be specific to the format that the schema describes (e.g. SLS, Flash, SVC). In order to use this metadata in a semantic-aware delivery framework, it is necessary to be able to map from the format-specific structures exposed by the binary schema, to the set of format-independent metadata needed by the RDO algorithm being used. The list of metadata required will vary depending on the particular RDO algorithm being used, but will generally include items such as

- segmentation of the content into Data Units;
- decoding interdependencies between Data Units; and

³It should also be noted that BSDL allows the bitstream to be described at whatever level is required, in order to avoid unnecessary verbosity. That is, if the reasoning to be performed requires only that the binary data be split into frames, then the BSDL Schema may be written in such a way that it emits a single XML element per frame. On the other hand, if certain fields within a frame are necessary for reasoning (such as a timestamp, sample rate, etc.) then the schema is able to expose these fields without showing the entire detail of the inside of a frame. See [95] for more information.
temporal relations between Data Units;

One such set of mapping rules may be used to describe the extraction of RDO metadata from SLS bitstreams, while another set of mappings describe the process for Flash, and a third for H.264/SVC (as shown – Figure 6.8).

A number of options exist for expressing such rules:

- **In the binary domain**: A BSDL Schema may be extended so that it appends attributes to the output which correspond to the needed RDO content features (size, dependencies, etc). The advantage of this approach is that the description of how these features are extracted from the binary data is very concise. The disadvantage is that this description is embedded in the BSDL Schema and is therefore tightly coupled, limiting reusability.
Listing 6.1 is an example of this approach. It shows part of a BSDL schema that outputs an rdo:ancestors attribute expressing the interdependency of Data Units. Thus, attribute declarations like this are one way to provide mapping rules from the format-specific binary structure of the schema, to the format-independent concepts needed for RDO (which are represented by members of the rdo namespace).

In the XML domain: A second option is to describe the identification of the necessary features using XQuery [49] or an XSLT [47] stylesheet. This removes the tight coupling with the BSDL Schema, but is less succinct, and adds an additional layer of complexity to the process. Listing 6.2 shows a fragment of an XSLT stylesheet that adds the same rdo:ancestors attribute to

---

The for structure used by the attribute value specification is not a loop, but rather a workaround for the fact that XPath does not have a current() function (cf. the sps variable in Listing 6.2).
What are you trying to say?

In the semantic domain: Alternatively, the BSDL Schema may be directly converted to OWL classes, allowing the feature identification process to be specified using an ontological reasoning tool such as the Semantic Web Rule Language (SWRL) [156]. One disadvantage of this approach is that RDF is inherently unordered, and so Data Unit order must be explicitly imposed using sequence numbers, timestamps or the like. Furthermore, some assertions about the order of such sequence numbers are non-monotonic (see for example Listing 6.4).

Examples of mapping rules using SWRL are (the prefix svc is used for the BSDL Schema, and rdo for the RDO ontology):

\[
\text{svc:nalUnit}(?x) \rightarrow \text{rdo:dataUnit}(?x)
\]

\[\ldots (6.3)\]

which implies that a Network Abstraction Layer (NAL) Unit is an atomic unit of data for the purposes of RDO. This deceptively simple rule is in fact making use of the inheritance properties afforded by SWRL and the semantic web, since there are no direct instances of \text{svc:nalUnit} within an \text{svc} instance, but

Listing 6.2: XSLT fragment annotating \textit{pps} elements with ancestor metadata

\[
\begin{xsl:template}
\begin{xsl:variable}
\begin{xsl:attribute}
\begin{xsl:template>
\]

a BSDL description.
What are you trying to say?

\[
\text{svc:pps(?pps)} \land \text{svc:spsID(?pps,?spsID)} \land \\
\text{svc:sps(?sps)} \land \text{svc:spsID(?sps,?spsID)} \land \\
\text{svc:seqNo(?pps,?ppsSeqNo)} \land \text{svc:seqNo(?sps,?spsSeqNo)} \land \\
\text{swrlb:lessThan(?spsSeqNo,?ppsSeqNo)} \\
\rightarrow \text{rdo:dependsOn(?pps,?sps)}
\]

**Listing 6.4:** A not-quite-complete rule for specifying the interdependency between a PPS and the SPS it references

rather it is the abstract superclass of all other top-level objects in an SVC stream. This inheritance is unavailable to an XSLT-based rule (e.g. Listing 6.2 on the preceding page), where separate rules must be specified for each instance type); and Listing 6.4 which (almost) states that a Picture Parameter Set (PPS) has a dependency to the *most recent SPS with an ID that matches the one given in the PPS*. If multiple SPS' with the given ID are present in the bitstream prior to the PPS, then rule 6.4 will incorrectly match all of them.

The missing constraint—"most recent"—is nonmonotonic and hence not supported by SWRL or OWL-DL\(^5\). Consequently, an XML-based approach has been applied to the mapping rules for syntactic parameters in the example system implemented in this work (see Section 6.4 on page 190). Future work on SWRL and/or other Semantic Web rule languages may provide the expressivity needed for this and other rules required for RDO parameters.

Examples of mapping rules for \(\Delta_{rate}\) and \(\Delta_{time}\) are given in Section 6.4 on page 190.

---

\(^5\)The missing “most recent” constraint is specified in the XSLT example (Listing 6.2) by the preceding:: axis and [1] predicate
6.3.2.2 Semantic Analysis

The aim of semantic analysis is to generate metadata that can subsequently be reasoned on to compute Semantic Distortion. There are many options for generating and obtaining semantic metadata, as discussed below. Crucially, there are also several disparate widely-used methods for serialising this metadata (RDF [111], XML [37], as well as numerous binary forms [71, 76, 148]). To be able to reason on such metadata (in order to compute Semantic Distortion) it must generally be translated into a single form. There are several options for this, but for the sake of brevity, and because it is the most powerful option for semantic reasoning, this section will focus on translation of binary and XML metadata into RDF/OWL.

Generating the desired content semantics

The first stage of semantic analysis involves extracting the desired semantics from the content (e.g. “this scene depicts the studio anchor discussing the news story”—see Figure 6.10 on page 184). As described in the introduction, this remains a challenging problem, with many efforts directed toward algorithms able to expose various specific semantics for media content. This process typically uses the uncompressed samples of the original content and (partly because of the volume of this data) can be very computationally expensive (Figure 6.6 on page 166). While such semantic metadata may not be specifically designed for the delivery process, it can often, nonetheless, contribute to it. For either of these reasons the semantic analysis may
often be performed asynchronously to the operation of the RDO-hinter. Further, much semantic metadata is presently annotated by hand, consider Flickr/Youtube tags, or iTunes song ratings, for example.

Whether semantic metadata is the result of an (a)synchronous analysis step or manual annotation, the result is a set of metadata about the content that is expressed in some machine-processable form. Such metadata is increasingly specified using ontologies, which simplify the integration of heterogeneous data sources as well as the reuse of information for applications other than those for which it was first developed [157]. However, this is far from universal, and a great deal of existing semantic metadata is stored as XML or binary data (see, for example, [71, 76, 148]).

Translating XML metadata

As discussed earlier (Section 6.3.2.1), it is possible to map XML-Schema-based metadata directly into the semantic domain [155]. This may be imprecise—it is generally possible to express the same semantics using several different XML structures (e.g. element content vs attributes)\(^6\). An alternative proposed by Hunter is to use an upper-level ontology [112], which is more robust, specifically because it involves a time-consuming manual mapping from the (implicit) semantics underlying the XML representation to a set of explicit ontological relations. Both of these approaches are

---

\(^6\)In contrast, when Section 6.3.2.1 discussed mapping BSDL Schema to an ontology, there was no such imprecision. BSDL has already restricted the expressivity of XML Schema in order to guarantee an unambiguous mapping between the binary and XML domain and back again. As a result, mapping BSDL into the semantic domain is also unambiguous.
feasible for a semantic-aware delivery framework.

**Translating binary metadata**

There are a number of very widely used binary formats for semantic metadata: ID3 [76], EXIF [138], and MPEG-4/Quicktime [5, 153] for instance. In this case, a syntactic analysis of this metadata must precede further processing of the semantics themselves. This syntactic analysis can be performed in the same manner that interdependencies and other constructs are exposed (Section 6.3.2.1). This will yield an XML description of the structure of the metadata, including the name and value of all of the desired metadata fields. This XML representation may then be mapped into the semantic domain, as above.

**Computation of Semantic Distortion**

This is the second stage of semantic analysis, and is one of the central contributions of this work. *Semantic Distortion* (SD) is defined as a measure of the “SNR” between the intended semantic (meaning) of the content before it is encoded, as compared to the semantics conveyed by the content that is rendered for its recipient(s). The contribution to Semantic Distortion that this chapter is primarily concerned with is that contributed by the delivery process, however the approach may also potentially be useful for other aspects of multimedia processing.
Clearly, this notion of Semantic Distortion is highly subjective (as indeed are many of the semantics of any given piece of media content). However, even approximations of Semantic Distortion as perceived by parties on the server-side of the process possess substantial value for optimising the delivery of the content semantics, as shown in Section 6.4 on page 190.

Given this definition of Semantic Distortion, it is possible to define a series of rules that map from concepts expressed in semantic metadata to a quantitative measure of SD. Although the content of mapping rules for SD will differ from those of syntactic analysis (see 6.3.2.1 on page 167), they have the same range of options for specification: directly within a (binary) schema, in the XML domain, or in the ontological domain. In contrast to the aforementioned syntactic mappings, SD rules translate readily into SWRL, such as Listing 6.5 which states that if there is an instance of

---

**Listing 6.5**: A SWRL rule that specifies the Semantic Distortion of English Communication

---

where **cyc**: refers to the CYC upper-level ontology [158], **time**: to the OWL-Time ontology [159], **dolce**: to the DOLCE ontology [160], and **rdo**: to the ontology defined in 6.3.3 on page 180.
Communicating during a certain time interval that uses the English Language, then the magnitude of the Semantic Distortion for that interval is doubled\(^8\). This rule covers both spoken communication (in which case the SD is associated with the audio track(s)), and visual communication (e.g. subtitles; where the SD is applied to the video).

**Combination of Semantic Distortion with sample distortion**

This is pivotal to the correct operation of the R-D optimisation algorithm. Chou [101] considers Sample Distortion to be *additive*, that is, the overall distortion \(D(\pi)\) is a large initial value \(D_0\) less the sum of the distortion of all packets \(D_l\) *received and useful* (which are computed by the product sequence):\(^9\)

\[
D(\pi) = D_0 - \sum_l \Delta D_l \prod_{l' \preceq l} (1 - \epsilon(\pi_{l'}))
\]  \hspace{1cm} \ldots (6.6)

However, the sample distortions used in Equation 6.6 are all measured according to a single algorithm, and hence have the same scaling and are directly comparable. This is not usually the case for Semantic Distortion, and is certainly not so when comparing Semantic Distortion with Sample Distortion. Instead, it is proposed that Semantic Distortion be considered to be *multiplicative*; that is, that SD represents a

---

\(^8\)Note that the factor of 2 is in this case relatively arbitrary—yet still shown to be useful (Section 6.4)—see Section 7.1 on page 207 for a discussion about possible future work on methods for evaluating Semantic Distortion.

\(^9\)where \(l' \preceq l\) selects all packets \(l'\) that are ancestors of \(l\) as well as \(l\) itself, \(\pi\) is the vector of packet transmission policies, and \(\epsilon\) the error/delay probability distribution for any given policy.
weighting factor that may be applied to a value of sample distortion for a packet, or group of packets. There are several motivations for this:

- First, multiplicative combination obviates the need for normalisation based on potentially unknown response curves for distortion algorithms (both sample and semantic). For example, say a Data Unit has a Sample Distortion with a magnitude of 0.3 dB, and a Semantic Distortion (based on the language of the communication) of magnitude 2. It is clear that these values cannot be combined additively without ensuring that they are first normalised to the same scale. However, while sample distortion uses objective measurements such as (P)SNR, the same cannot in general be said of subjective measurements of Semantic Distortion. Even though there are numerous quantitative measures (see for example [161]), comparison between data from different quantitative tests must be done very carefully. Furthermore, Semantic Distortion is intended to encompass a wide range of data, as has been discussed (Section 6.1 on page 155), beyond formal subjective testing. Combining disparate data sets multiplicitively avoids this need to normalise.

- Combination of several Semantic Distortion data-sets relating to a piece of content typically has similar issues relating to normalisation. Consider, with the above example, the addition of a second Semantic Distortion data set computed from Temporal Information (TI). The SD for the Data Unit in question has a magnitude of 0.7. Combining this datum with the others is straightforward if it is considered to be a scaling factor (i.e. multiplicative).
Finally, multiplicative combination retains a known zero point. This is important if either sample or any Semantic Distortion has a magnitude of zero; in the first case, this indicates that the packet has no effect on the reconstruction of the signal; in the second, that it does not convey any semantics. Either way, these features must be transmitted to the output distortion value.

### 6.3.3 An Ontology for Semantic Distortion

The mapping process described in Section 6.3.2.2 on page 174 requires the definition of appropriate concepts to be used as the destination of the rules. These concepts fall into two categories: the formal definition of a Data Unit in so far as it pertains to R-D optimisation, and the definition of Semantic Distortion itself. These are described below in Sections 6.3.3.1 and 6.3.3.2, respectively. These definitions and their associated concepts are attached to the DOLCE [160] upper-level ontology, because of its precise separation of fundamental concepts\(^\text{10}\). Figure 6.9 on page 183 depicts the Semantic Distortion ontology (prefixed by sd) along with its DOLCE ancestors (prefixed by dolce). Refer to [160] for a full treatment of DOLCE; the following description should suffice with regard to the present work.

The fundamental distinction in DOLCE is between *enduring* and *perduring* entities (Figure 6.9). The precise philosophical definition of these terms is complex and also somewhat controversial [160], but for the purpose of this chapter it will suffice to say that the former are entities that *exist* in some region of time (and possibly

\(^\text{10}\) As described previously (Section 6.3.2.2 on page 174), the choice of DOLCE is not normative but rather a preferred embodiment.
space), whereas the latter are events that *occur* during a region of (space-)time. Both endurants and perdurants have *qualities*, and a distinction is made between a quality (e.g. color, temperature) and its *quale*—a region defining the “value space” of a particular quality (e.g. red, 298K). This is partly inspired by the fact that an endurant individual will permanently have particular quality individuals (i.e. it will always have a color), but the value of those qualities may change over time. Quales belong to the class of all *abstract* concepts that are neither endurant nor perdurant.

While DOLCE includes the abstract notions of a temporal quality and a temporal region, RDO requires a more concrete conceptualisation of time in order to be able to synchronise semantic metadata with the underlying media Data Units. Furthermore, the metadata that a semantic-aware delivery framework must assimilate will have a large variety of fundamentally different representations of time:

- MPEG-7 [162];
- SMPTE [163];
- XML Schema [164];
- OWL Time [165]; as well as
- the innumerable binary syntaxes used by media formats.

Each representation uses a different syntax to represent time. If these are to be reasoned on as part of semantic-aware delivery, methods are required to translate from one representation to another.
Throughout the proceeding discussion, the example shown in Figure 6.10 on page 184\(^1\) will be used to illustrate each concept. The example consists of a short Audio-Visual clip that forms part of a news article on events in the Middle-East\(^2\). The first part of the clip depicts a studio presenter introducing the story (the temporal interval containing this section is described by an owlT:DateTimeInterval, and the visual and aural communication features with cyc:(Visual/Audio)Communicating). Subsequently, contextual footage is shown of the subject walking to the podium while an off-screen narrator continues the story. Finally, the subject speaks in Persian with English subtitles appearing below (using similar owlT:DateTimeInterval and cyc:(Visual/Audio)Communicating instances). These features are annotated via CYC [158] classes and properties and then reasoned on using mapping rules (Listing 6.4 on page 173).

---

\(^1\)Individual IDs used in Figures 6.10–6.12 are used purely to differentiate individuals from each other. The general naming scheme used for these IDs is to abbreviate the type name of the individual and append a number which increments from 0, 1...n for each type. For example, the first instance of the Owl-Time class DateTimeInterval in Figure 6.10 has the ID dti0.

\(^2\)This clip was used by permission from SBS World News Australia.
Figure 6.9: An ontology for Semantic Distortion
Figure 6.10: Example of the semantic annotation of an Audio-Visual Clip
6.3.3.1 Data Units

For the purposes of R-D Optimisation, Chou designates an atomic segment of data as a DataUnit [101], where each packet on the network may contain at most one Data Unit. As shown in Figure 6.11, individual Data Units are composed by one or more Extents (contiguous sections of memory), which may be located by an hasOffset and hasSize properties. The range of both properties is LogicalQuantity, which in turn is defined by an atomic size (another LogicalQuantity) and a magnitude. Two LogicalQuantity individuals are defined by the ontology (bit and byte) for use
by the `hasAtom` property, but it may use any other logical quantity as required. Alternatively, an extent may be located via a size and an immediate relation to a predecessor (the inverse of which is `hasImmediateSuccessor`—not shown), or by a sequence number (e.g. when referring to a Data Unit in an RTP packet).

Rule 6.3 on page 172 is an example of the use of `DataUnit`, showing how it enables format-independence by mapping from arbitrary format-based structures to a single interface for RDO. Figure 6.11 on the previous page depicts the example H.264/AVC bitstream previously discussed in Chapter 3 on page 63, along with instances of the Semantic Distortion classes that deliniate the Data Units in the stream.

The first Data Unit in Figure 6.11 (du0) is specified by a single extent with `hasOffset` and `hasSize` properties (that is, the byte offset of the first byte within the Data Unit, as well as its size in bytes). Subsequent Data Units are specified as immediately subsequent to their predecessor, and having a certain size. This mode of specification is semantically equivalent to explicitly specifying the offset and size of every Data Unit.
Figure 6.12: Example instances of Semantic Distortion classes describing the distortion of a H.264/AVC bitstream

6.3.3.2 Distortion

Distortion is the other central concept in RDO. It is a type of Logical Quantity that measures “the amount by which the distortion at the receiver will decrease if the Data
Unit is decoded (on time)” [101]. Distortion has at least two distinct types: sample and semantic, which were described earlier (Section 6.3.2.2 on page 174). Distortion in general has a continuous value-space (continuousQuale), which in turn has a data-type property hasMagnitude. Distortion also has an hasAggregativeBehaviour property (this relation is not shown), which may be Additive or Multiplicative. It is left to the user to decide which behaviour(s) to assign.

Figure 6.12 on the preceding page shows an example of the distortion instances for a bitstream. In this example, each NAL unit in a H.264 bitstream is given a sample-based distortion according to its contribution to a scheme (for example that proposed by Chou [101])—these are the DataUnit, Distortion and Quale individuals toward the top of the figure. In this instance, SemanticDistortion applies to more coarsely-grained Data Units. These are specified using OWL-Time intervals (by mapping from the intervals previously shown in Figure 6.10 on page 184). Each instance of Semantic Distortion intersects many Data Units in the actual media. Assigning Semantic Distortion to individual media Data Units is done by the mapping rules associated with syntactic analysis (Section 6.3.2.1, above).

6.3.4 Summary

In summary (refer to Figure 6.1 on page 155), there are three primary components to the proposed semantic-aware multimedia delivery framework:

- a hinter, which computes all of the content-based metadata offline;
a delivery node which is left with as little work to do as possible, by virtue of the hinter having already performed much of the necessary computation. The delivery node simply forwards or drops each packet as it arrives, on the basis of some optimisation algorithm; and

- semantic analysis, used to provide the hinter with metadata that can be used to compute Semantic Distortion.

The novelty of this work consists in tying semantic analysis and semantic metadata to the R-D hinter, as well as an architecture for this hinter so that it operates in a format-independent manner. More specifically, the work contributes:

(a) the definition of Semantic Distortion (SD) (Section 6.3.2.2 on page 174);

(b) an ontology for RDO concepts and for SD that enables it to be inferred from arbitrary semantics, and then combined with sample distortion (Section 6.3.3 on page 180);

(c) extension of the concept of an R-D Hinter to encompass SD (Section 6.3.2 on page 165);

(d) a format-independent architecture for the semantic hinter, which operates by extracting all format-specific details into declarative data (schemata and mapping rules). It is argued that this is imperative to allow the increasingly diverse range of formats and devices to interoperate (Section 6.3.2.1 on page 167); and
(e) a semantic-independent architecture for the hinter, again accomplished by using schemata (ontologies) and mapping rules (Section 6.3.2.2 on page 174);

6.4 Subjective Testing

6.4.1 Methodology

Double-blind, randomised subjective testing was used to validate the hypothesis that the use of Semantic Distortion can improve multimedia delivery. The scenario used for these tests was a mobile environment—where channel characteristics are often highly variable, and handset capabilities mean that audio and video require relatively similar bandwidth. As such, the source material was encoded at 22.05kHz for the audio, and the video at QVGA resolution and 15 frames per second. Initial trials were conducted using a mobile (cellular) handset, but it was decided that this introduced a significant number of variables (e.g. the particularly small screen size, problems with controlling playback, and uncertainties about the quality of the audio rendering hardware) without lending any additional credence to the experiment per se (as opposed to conducting the trials using a notebook, but using mobile-ready content). Consequently, respondents evaluated video displayed on the screen of a Compaq nc4000 notebook (1024x768 total resolution, 12” screen), and listened through Sony MDR-V500 headphones. Respondents were free to adjust volume and viewing distance as desired, with the latter ranging from 8 to 16H (the QVGA image measured 75mm W × 58mm H). The testing was conducted according to ITU-T P.911 [161], includ-
ns prescribed in Table 4\textsuperscript{13}. Pairwise Comparison (PC) was used to test the hypothesis that

\textit{Semantic Distortion in multimedia delivery improves the communication of the meaning/semantics of the content.}

nineteen respondents were asked to decide which clip (A or B) “best represents the news article to you.” The complete written instructions given to participants can be found in Appendix I, Figure I.1. Respondents were not experts of multimedia delivery, or subjective testing. An approximately equal number of each gender was chosen, and participants ranged in age from 16 to 24. Respondents had normal or corrected-to-normal eyesight, and (with the exception of two participants who had mild age-related hearing loss).

There were four clips plus an initial (hidden) training clip, all exhibiting some or all of the characteristics depicted in Figure 6.3 on page 158. Three were news footage, and the fourth part of an interview between an English interviewer and a Japanese interviewee, all between 25 and 45 seconds in length. The audio from each clip was encoded using Scalable to Lossless Coding (SLS) [120] with an AAC base layer of 6kbps to provide a large scalable range. Scalable Video Coding (SVC) [15] was used for the video with 8 coarse-grained scalability (CGS) SNR (quality) layers (with

\textsuperscript{13}screen luminance, ratios & chromitacity, background illumination and noise level
Israel
A news article regarding a resumption of bilateral peace talks between Mahmood Abbas, president of the Palestinian Territories, and Ehud Olmert, Prime Minister of Israel.
(Courtesy SBS News Australia)

Haneef
Mohammed Haneef was arrested, deported and later exonerated by the Australian Federal Police, in relation to links with relatives responsible for the failed bombing at Glasgow airport.
(Courtesy SBS News Australia)

Kunoichi
A short excerpt from an interview with Masahiro Kumono, lead developer of the Sega videogame, Kunoichi.
(Public Domain)

Iran
Excerpts from a press conference by Mahmood Ahmadinejad, president of Iran, welcoming a US report that Iran had halted its nuclear weapons program, and a sceptical response from Ehud Olmert.
(Courtesy SBS News Australia)

Figure 6.13: The test corpus consists of four short video clips that were both semantically-adapted and bitrate-adapted to various target rates

LQP at 30, 34, 38, 42, 45, 48, 51, 54 for layer 0 to 7 (respectively), and \( RQP = LQP + 2dB \) and 4 medium-grained SNR layers. Spatial and Temporal layers can be beneficial to semantic-aware optimisation (see, for example Cranley [104]) but it was decided to limit the sources of variability for the present experiment. In that regard, no attempts were made at error concealment\(^\text{14}\), even though this would have an impact on a user’s perception of a real world system employing SD.

\(^{14}\)except for silencing of an SLS decoder bug observed at particularly low bit rates that that led to saturation of the signal in sections where there should be silence. This bug was observed equally on both clips in a pair, and would otherwise have caused discomfort for test subjects.
What are you trying to say?

\[
\text{cyc:VisualCommunicating}(\text{?video}) \land \text{t:DateTimeInterval}(\text{?dti}) \land \\
\text{dolce:HAPPENS--AT}(\text{?video}, \text{?dti}) \land \text{cyc:imageDepicts}(\text{?video}, \\
\text{cyc:StillImage}) \land \text{swrlx:makeOWLThing}(\text{?sd}, \text{?dti}) \land \\
\text{swrlx:makeOWLThing}(\text{?sdq}, \text{?dti})
\]
\[
\rightarrow \\
\text{rdo:hasDistortion}(\text{?video}, \text{?sd}) \land \text{rdo:SemanticDistortion}(\text{?sd}) \land \\
\text{dolce:HAS--QUALE}(\text{?sd}, \text{rdo:Multiplicative}) \land \text{dolce:HAS--QUALE}(\text{?sd}, \\
\text{?sdq}) \land \text{rdo:ContinuousQuale}(\text{?sdq}) \land \text{rdo:hasMagnitude}(\text{?sdq}, 0.5)
\]

**Listing 6.7**: A semantic rule from the test data asserting that still images have a (relative) SD of 0.5

### 6.4.1.1 Semantic analysis

Semantic analysis for each clip was conducted using classes from the Cyc [158] ontology, to provide semantics indicating the language of communication (spoken or written), among other things. The choice of Cyc for this task was purely as an example, the semantic-aware delivery framework places no constraints on specific metadata ontology(s) (as discussed in Section 6.2 on page 158). Mapping rules were created for these classes (Listing 6.4 is one of these, and Listing 6.7 another\(^{15}\)) to describe how particular semantics relate to SD. Temporal regions were specified using OWL-Time [165]. Again, this choice is by way of example only, other temporal schemata may equally be used. Figure 6.12 on page 187 depicts example SD and OWL-Time instances.

\(^{15}\)Full rules are available in Appendix I
Figure 6.14: Semantic adaptation diverts more bits to the portion of the content containing more of the meaning.
6.4.1.2 Syntactic analysis

Syntactic analysis was conducted using a BSDL Schema for SLS and another for SVC (see Appendix I), then an XSLT stylesheet to map SLS & SVC fields to the necessary RDO metadata (as per Section 6.3.2.1 on page 167). Delivery optimisation was performed using a very simple algorithm, so as to limit (as much as possible) the testing to the Semantic Distortion concept, rather than introduce a second independent variable in a sophisticated optimisation routine. Essentially, the algorithm used was

1. Using the rules generated in Section 6.4.1.1, compute all of the SD values for the entire clip;

2. Aggregate the SD values separately for audio and for video, according to the behavior specified (see Sections 6.3.2.2 and 6.3.3);

3. Segment the clip into regions so that each region has a constant SD for audio and a constant SD for video;

4. For each region

   (a) Apportion the target bandwidth between the audio and video stream according to the aggregated SD for each component;

   (b) Truncate each SLS frame so as to achieve the apportioned audio bit-rate; and

   (c) Drop SVC NALUs to most closely approximate the target video rate
(while respecting the discardable flag).

Each clip was encoded to three different bit rates using this method, for a total of twelve clips, plus the hidden training clip. For each semantic-aware clip produced using this algorithm, a reference clip was created with the same average audio and video bit rates as the semantic-aware clip (by truncating the SLS and dropping SVC NALUs). This means that the semantic-aware clip devotes more of the available bandwidth to that part (in this example, audio or video) that carries more of the semantics of the content, whereas the reference sample uses the same total bandwidth, but has a static ratio between audio and video. This is illustrated in Figure 6.14 which shows the semantically-adapted and equivalent average rate series for the audio tracks of the high-bitrate “iran” sequence. The video tracks are not shown since the coarser granularity of the video scalability means that variance is too great to discern average trends. Nonetheless, the audio tracks clearly show how the adaptation algorithm responds to varying SD, and also that both audio tracks have the same total average rate.
Figure 6.15: Subjective testing shows a 72% preference for Semantically-aware multimedia delivery.
6.4.2 Results

In total, 72% of the semantic-aware clips were preferred by subjects when compared to the average-rate reference clip (as shown, Figure 6.15), with a variance of 20.57% and a 95% confidence interval of $\pm 5.74\%$.

Of the twelve pairs, one very low-rate semantic-aware clip was rated as worse than its average-rate partner. It is likely that this is due to the deliberate simplicity of the adaptation algorithm. A more sophisticated algorithm would be expected to deal with such outliers more effectively. Having said this, three respondents independently remarked that they preferred one particular low-rate non-semantic-aware clip because it accorded the speaker “more respect” by making his voice clear, even though they couldn’t understand it. Because the tests were double-blind, it is not known whether this comment corresponded to the clip in question.

Another two clips were voted as no better and no worse, and the remaining nine semantic-aware clips were preferred 84% of the time. This demonstrates that Semantic Distortion is of significant benefit in the multimedia delivery process. Moreover, the system proposed in Section 6.3 is effective in processing Semantic Distortion and R-D optimisation-related metadata in a way that meets the objectives identified in Section 6.2. In contrast, however, the result also suggests that the use of Semantic Distortion to optimise the apportionment of bandwidth between audio and video streams could possibly not be beneficial for a minority of content, at least without

---

16 variance is not considered to be particularly informative in this instance, due to the binary nature of each sample in a Parwise Comparison (the respondent picks one clip or the other). Because of this, every sample is relatively distant from the mean.
more sophisticated optimisation algorithms. However, while the modal trade-offs employed for these few cases fails to yield an improvement, it is quite possible that other uses of Semantic Distortion (see Section 6.3.1 on page 162) may give the desired results. Further investigation of this is left to future work.

6.5 Conclusion

This chapter describes a framework for incorporating semantics into the multimedia delivery process. It builds on existing work for exposing semantics in content and delivering media in a rate-distortion optimal way. In effect, this alters the conceptual end-points of the multimedia delivery chain. Instead of server-client, using semantics extends the process to (human) creator-consumer, by minimising distortion of the \textit{intended meaning} of the content (see for example the news report in Figure 6.3 on page 158). At the same time, the framework provides the flexibility to incorporate new semantics, optimisation algorithms, and content formats as they become relevant. This process can operate largely without the addition of new software or hardware components, since format-specific details are provided in schemata rather than hard-coded.

The framework has been validated via subjective testing that asked candidates to make a pairwise comparison between a video clip that had been semantically adapted (more bandwidth devoted to that mode carrying more of the content semantics) and one adapted to an equivalent constant average bitrate. In total, 72\% of the seman-
tically adapted clips were preferred by subjects when compared to the average-rate reference clip. Of the twelve pairs, one semantically adapted clip was rated as worse than its average-rate partner. Another two were voted as no better and no worse, and the remaining nine semantically adapted clips were preferred 84% of the time.

This result demonstrates that Semantic Distortion is of significant benefit in the multimedia delivery process. Furthermore, it validates the format-independent architecture proposed in Figure 6.4, as well as the simple algorithm used to semantically adapt content along its modal axis, across a range of bit-rates typical of current mobile communication channels. Nevertheless, the results suggest significant scope to develop higher performance semantic adaptation algorithms. Some possibilities for this are suggested in the following chapter.
Chapter 7

Scattered to the winds

Conclusions and Future work

[ In the Matrix ]

Neo: (looking at a helicopter) Can you fly that thing?
Trinity: Not yet. (takes out cellular phone)

[ The real world ]

Tank: Operator.
(Trinity – voiceover): Tank, I need a pilot program for a V-212 helicopter. Hurry....

(Images of helicopter and Trinity’s head/brain on the computer terminal operated by Tank).

[ In the Matrix ]

(Trinity’s face twitches as if from an electric impulse)

Trinity: Let’s go.

(Cut to shot of Trinity flying the helicopter and Neo wielding a large cannon attached to its side).

— The Matrix, 1999, Warner Bros. USA [166]
In the 1999 science-fiction film “The Matrix” [166], characters were able to acquire any new skill (such as martial-arts or helicopter piloting) by downloading this knowledge as data files directly into their memory. This was made possible by creating an abstraction of a human being similar to that of the systems proposed by this thesis. The hardware (the body), as well as the software (the mind) were kept generic (that is, format-independent) to allow the system to adapt to unfamiliar circumstances, simply by downloading new data. This is how the proposed systems are able to respond to previously unknown semantics, media or metadata formats. Certainly, the present work has less auspicious aims, but, nonetheless, this approach allows both characters in the film and the work in this thesis to easily expand to accommodate future requirements.

In the latter case, this is motivated by the observation that the diversity of the multimedia landscape (content providers, formats, deliverers, processors, channels, consumption devices—Figure 7.1) is growing exponentially. Thus, maintaining interoperability between elements of this landscape as new developments are made (say a new content format is released) is increasingly challenging. In general, hardware and software vendors for particular components must all (largely independently) develop new modules for their devices.

In contrast, this thesis proposes format-independent methods of performing various tasks in the multimedia delivery process, so that providing support for a new content format (for example) requires no changes to hardware or software, but simply dissemination of data files that describe how the new format is to be processed. The
core aspects of this work are as follows:

- The need for a multimedia syntax description mechanism that can cope with current and future multimedia formats;
- Means to describe how content may be fragmented for streaming or other forms of delivery;
- Extraction of temporal parameters for each content fragment;
- Generalisation of these tools to encompass binary, textual and XML multimedia metadata;
- Semantic Distortion, which measures the transmission of meaning from content creator to content consumer; and
Development of a framework to exploit Semantic Distortion in the media delivery process.

For multimedia, a fundamental requirement for format-independence is the ability to describe the structure of any given format in a way that exposes how it may be fragmented for delivery or processing, and how other data important to the processing (for instance temporal or scalability parameters) can be extracted from the binary data. Several meta-syntax languages are evaluated that (to greater or lesser degree) perform this function. Of these, the most suitable for general use in format-independent processors is found to be MPEG-21’s Bitstream Syntax Description Language (BSDL). Its general suitability notwithstanding, BSDL exhibits several critical flaws when used to describe and process modern content formats such as H.264/AVC, AAC, and SLS. Specifically, extensibility for complex data types is lacking, and the time required for processing increases quadratically relative to the duration of the bitstream. In response, this thesis proposes several new features for the language which reduce processing complexity to a linear function of bitstream duration and provide the required extensibility. These features are implemented and validated using bitstreams of real-world length, which exhibit a linear response of approximately ten times the speed of playback (on the particular test machine used), for videos up to one hour in duration.

This thesis has proposed a Bitstream Binding Language (BBL), which provides a flexible and format-independent mechanism for delivery of rich multimedia Content. The central contribution of BBL is a framework for the description of mappings
Scattered to the winds

between a rich media container and a delivery channel, which may be a streaming format such as RTP, or a static format (an ISO/QuickTime file, for example). BBL has been adopted by MPEG as part of the MPEG-21 Multimedia Framework, and can be used to describe such mappings for any multimedia content format, enabling the delivery of format-agnostic virtual containers, and facilitating interoperability with new media types as they are developed.

A BBL Processor has been developed, and its use demonstrated for MPEG-21 Digital Item delivery over MPEG-2 Transport Streams and RTP. In addition it has been shown how BBL can be used to provide a generic mechanism for translating the content from one syntax to another. Using the Music Player MAF format as an example, it was shown how BBL can extract fragments of binary and XML data and insert them into a different output format, so that existing devices are able to render the content. BBL can also act as a format-independent hinter and virtual container assembly mechanism.

In addition to providing a mapping for content into one or more output formats, BBL may be used to customise the presentation of content according to the requirements of the user, terminal and/or delivery channel. For example, BBL could be used to insert Supplemental Enhancement Information [12] into an H.264/AVC stream, for those channels/terminals which can make use of the extra information. Further, with a scalable bitstream, BBL could be configured so that it delivers different layers to different clients, depending on the parameters of each.

In the light of Reconfigurable Media Coding (Appendix A), the flexibility provided
by BBL becomes even more significant. In effect, this represents a paradigm shift away from highly rigid bitstream formats, to an approach that allows the bitstream to be completely restructured according to the requirements of the content, channel or terminal. Reconfigurable Media Coding provides a decoder that allows new video formats to be developed without extensive standardisation and application development efforts. BBL brings similar ease of extensibility to a multimedia streaming and delivery architecture.

Finally, Semantic Distortion (SD) is proposed as a measure of the “SNR” between the intended semantic (meaning) of the content before it is encoded, as compared to the semantics conveyed by the content that is rendered for its recipient(s). Utilising SD to optimise multimedia delivery effectively alters the conceptual end-points of the multimedia delivery chain: rather than server-client, using SD extends the process to (human) creator-consumer, by minimising distortion of the intended meaning of the content.

A framework is proposed for semantic-aware multimedia delivery using Semantic Distortion which provides the flexibility to incorporate new semantics, optimisation algorithms, and content formats as they become relevant. This process can operate largely without the addition of new software or hardware components, since format-specific details are provided in schemata rather than hard-coded. The framework has been validated via subjective testing that asked candidates to make a pairwise comparison between a video clip that had been semantically adapted (more bandwidth devoted to that mode carrying more of the content semantics) and one adapted to an
equivalent constant average bitrate. In total, 72% of the semantically adapted clips were preferred by subjects when compared to the average-rate reference clip. Of the twelve pairs, one semantically adapted clip was rated as worse than its average-rate partner. Another two were voted as no better and no worse, and the remaining nine semantically adapted clips were preferred 84% of the time. This demonstrates that Semantic Distortion is of significant benefit in the multimedia delivery process.

7.1 Future Work

There are numerous avenues for extension of the work in this thesis. Principal among these is continued efforts toward fully Reconfigurable Media Coding (RMC) as described in Appendix A. Contributions to RMC are proposed there, along with a discussion of future work in that area.

This section proposes further work that is suggested by the material contained in previous chapters. Firstly, extensions to a system that uses BSDL are considered that can obviate the primary source of complexity in using BSDL to parse binary data: generating equivalent XML data. Secondly, this section describes several aspects of the Semantic-aware delivery framework that merit further consideration, including declarative methods for describing semantic algorithms, and other forms of semantic metadata that may be used to further optimise the semantic delivery process.

The application proposed for BSDL by its authors (see Chapter 3) was to parse binary content and generate XML metadata. In Chapter 4, it is used purely as a schema for
the binary content itself, and although XML is used as an abstraction in this process, it is not actually necessary, or desirable\(^1\), for the XML to actually be produced. BiM (discussed in Section 2.3.2) addresses a more constrained version of this problem, by developing a parser that takes a BiM binary stream as input, but has an XML API (such as SAX or DOM) as an output. Generalising this approach to provide a parser which can operate on arbitrary binary input (described by a BSDL schema), and provide an XML or other type of API as output would be valuable future work, removing much of the computational complexity associated with the BBL process.

The primary challenge involved with this task is reverse-engineering XPath (or a subset of it) into some sort of BPath (Binary Path language) which takes an XPath expression and applies it to a binary document. As suggested in Chapter 4, these intermediate BPath expressions could be compiled into a byte-code version of the BBL document that is then more efficient when processing many content instances. Achieving a BPath representation for simple path expressions (such as `/MPEG4/VOP[32]/Startcode`) is straightforward, because the BSDL schema clearly links the structure of the binary document to such hierarchical constructs. However, XPath is a very expressive language and the reverse-engineering task is likely to be much more complicated in many instances.

The semantic-aware delivery framework proposed in Chapter 6 focused predominantly on the format-independent semantic hinter. Future work may consider more closely the design of the semantic analysis and delivery node modules (see Figure 6.4

\(^1\)because of the significant computational complexity involved in producing verbose, textual, XML.
Scattered to the winds

on page 161). In this work, the syntax of compressed media content was described declaratively (using schemata) to enable a generic hinter to extract data for the RDO process. Semantic analysis, on the other hand, is generally conducted using raw (uncompressed) media data, however here too numerous content formats are used. Furthermore, there are a wide range of low-level semantic features (e.g. color, texture, luminance) that are extracted from the raw data in order to infer higher-level semantics. Future work could therefore investigate declarative mechanisms for (a) describing how low-level features are computed from any given content format, and (b) mapping low-level features to high-level semantics.

Secondly, future work should consider how to describe an R-D optimisation algorithm using declarative language. This chapter has addressed the format-independent design of the RDO metadata and send/drop module (Figure 6.5 on page 162). However, it has not fully addressed a generic mechanism for describing the RDO algorithm itself. Such a mechanism would allow new RDO algorithms to be installed in a diverse variety of existing delivery nodes without requiring their hardware or software to be upgraded.

Semantic Distortion and the other concepts introduced in Chapter 6 on page 153 suggest numerous avenues for future work. There is significant recent interest in the success with which Flickr, YouTube and other sites have employed Collective Intelligence (sometimes referred to as Folksonomy). With regard to SD, we believe that this sort of approach to authoring mapping rules could help to refine their accuracy. This is, however, a challenging problem, because mapping rules that work well in
one genre (for example) may not be appropriate for another, and also because even
with high-/level ontologies such as Cyc, there remain many different ways to assert
the same semantics, giving rise to a nontrivial aggregation problem (although this
problem is by no means unique to SD).

Furthermore, there is a great deal of metadata that accompanies multimedia such
as movies that may be very useful in determining SD. For example, future work
could develop techniques to utilise what the director or other artists involved in the
creation of a work have to say about its parts. In other words, given a commentary of
the type usually distributed with DVD films (or preferrably its transcript), how could
one identify statements about the relative significance of parts of the movie, and use
these to infer SD? The same could be said of the large amount of metadata generated
during the creation of content (which is not currently distributed). For example, if a
scene took 20 takes to get right, does that say something about its SD? Clearly this
sort of inference would not be emphatic, but it may nonetheless be useful as a guide.

Finally, just as users’ “clickstream” (the list of links visited when browsing the Inter-
net) contains a great deal of implicit semantics that are of significant value to adver-
tisers, the manner of consumption of multimedia content may also contain implicit
semantics relevant to SD. Examples of such semantics may be adjustments made to
volume levels, tracks or chapters skipped, or even measurements of the amount of
eye motion. Future work could investigate how these semantics can be exploited
within the SD-aware delivery framework.
Bibliography


http://www.ietf.org/rfc/rfc2250.txt?number=2250,  
1998.


[27] G. Stamou and S. Kollias, Eds., Multimedia content and the semantic Web:  

Intl. Conf. on. 2002, W3C.

Description,” Lecture Notes in Computer Science - Topic 16: Distributed  

representation,” in Multimedia and Expo, IEEE Intl. Conf. on, 2002, pp. 773–  
776.


[32] D Brickley and R.V. Guha, “RDF Vocabulary Description Language 1.0: RDF  


Appendix A

Tearing it all down again

Toward format-independent decoding

MEDIA coding has changed a lot since its infancy in the early nineties. The original MPEG video coding standard [13] was released in 1993, and since then MPEG-2 [8], MPEG-4 [167], AVC (Advanced Video Coding) [14] and now SVC (Scalable Video Coding) [15] have been produced.

Despite this growth in multimedia coding technologies, the process of standardising new algorithms and coding techniques remains very lengthy. This standardisation process has been necessary for coding technology to be useful to the wider public, because without it there is no guarantee that one vendor’s encoder will work with another’s decoder. The typical lead-time between innovation and mass-deployment for recent standards has been three to five years; work on H.264/AVC [12] for example began in 2002, but did not see significant use until the release of the video iPod in
late 2005 [168].

Each successive codec released by MPEG has been substantially more complex than the last, typically yielding twice the compression efficiency of its predecessor. Because of this growing complexity, the textual specification of recent standards (since MPEG-4 [9]) has increasingly been replaced by the reference software implementation as the true normative specification. However, while this normative specification (typically in C or C++) is very precise, it presents a number of limitations.

A large portion of compression technology (i.e. coding tools) are common across all MPEG standards, but there is no direct way to recognise this commonality. Additionally, the sequential C/C++ descriptions do not expose the potential parallelism that is intrinsic to the algorithms constituting the codecs. They have also become excessively large (in terms of lines of code) making it extremely labor intensive, for example, to transform the reference software into a VHDL\textsuperscript{1} implementation. In other words, the complex C/C++ specifications no longer constitute a good starting point for implementation of the standard. It would be preferable to develop formalisms that operate at a higher level of abstraction, that simplify top-down system development and design.

The large number of coding tools available also leads to difficulty in specifying predefined subsets for different application scenarios (i.e. standard profiles). As an example, a low complexity profile is often defined to provide the minimum configu-

\textsuperscript{1}VHSIC Hardware Description Language [169], a language that is used to describe how hardware circuits may be implemented in devices such as ASICs or FPGAs.
ration expected to achieve acceptable results on highly constrained decoding devices. However, specifying such profiles prior to, or soon after, release of the standard would not appear to allow the optimal combinations of tools to be identified. Furthermore, it is often not possible to identify all of the application scenarios in which a codec will be used, at the time of its release. Nor is it feasible to provide a normative profile for every scenario. Ideally, implementers of a standard should be able to select arbitrary combinations of the available tools, in the way that best matches the requirements of each application. The challenge with this approach is ensuring interoperability, and it is this aim that motivates the work described in this chapter: Reconfigurable Media Coding (RMC). RMC is a collaborative effort being undertaken by MPEG [116]; the contributions made specifically by this chapter are the Reconfigurable Syntax Description and Parser module. RMC provides a powerful mechanism for “freeing multimedia from hard-coded formats”, and this work builds on the contributions made in Chapter 3. It is included here (as an appendix) rather than in the body of the thesis, because of the considerable amount of background material that must be covered prior to the discussion of the actual contributions.

The following Sections consider the objectives (A.1) and usage scenarios (A.2) for a reconfigurable coding framework. After that, each of the components of RMC are discussed: the structure of an RMC bitstream (Section A.3), the CAL Actor Language (CAL) (A.4), and the framework as a whole (A.5). Finally, Sections A.6 and A.7 beginning on page A–15 present the central contributions of the chapter—the reconfigurable syntax description and parser generation, respectively.
A.1 Objectives

A recent trend in multimedia devices (Cell phones, music players, PDAs and the like) is convergence in terms of the functions supported on any single device. This means that the device must support an increasing number of media formats for images (such as JPEG or TIFF), audio (MP3, AAC, Real Audio) and/or video (MPEG-2, MPEG-4, AVC or Quicktime). Since many of these codecs share common or similar coding tools, the traditional codec-level conformance specification and implementation is not the most efficient way of implementing multiple codec support on real devices. However, this redundancy between coding formats is implicit, at best, and device vendors must expend a great deal of effort to identify and exploit this redundancy. The objective of an RMC framework is thus to describe current and future codecs in a way that makes commonality explicit, reducing the implementation burden for device vendors.

The framework has the following objectives:

- to create a flexible video and audio coding framework for new codec and coding tool development;

- to simplify the specification and adoption of new coding tools by explicitly reusing the desired elements of previous standards, instead of defining a new monolithic standard;

- to provide a new interoperable model of codec definition at the level of funda-
mental algorithmic modules (such as the Discrete Cosine Transform) that gives users the ability to utilise any module required to suit the requirements of the application, content or network; and

- to simplify the implementation process for new codecs by making component reuse explicit.

### A.2 Usage scenarios for reconfigurable coding

Media bitstreams can describe the decoders required to process them in several ways, which differ in the trade-offs they make with respect to, for instance, generality, processing requirements, openness, and infrastructure. For instance, in a library-based decoder, the bitstream describes its decoder as a network that consists of the instantiation (and parametrisation) of decoding tools taken from a library of predefined modules. This approach results in relatively small configuration overhead, but it assumes the existence of a standardised or otherwise agreed-upon library of decoder modules, or a mechanism by which a platform may acquire new modules on the fly (e.g. downloading them over a network).

At the other end of the spectrum are fully programmable decoders, in which the bitstream contains a complete executable description of its decoder in a platform independent specification. This scenario requires much more infrastructure on the decoder side, which needs to be able to quickly translate a decoder specification into an efficiently executable implementation on its specific hardware platform. It would
thus need to incorporate an complete compilation infrastructure for the decoder specification language. That language, in turn, needs to be platform-agnostic, and still yield reasonable implementations on a wide range of hardware and software targets.

In such a scenario, bitstreams may describe decoders that are arbitrarily tuned to their specific requirements, without a for encoder and decoder to agree on a specific library.

Somewhere between the two are hybrid decoders, in which some of the coding tools could be specified using an executable language, while others are instantiated from a standard library. A plausible instance of this would be the executable description of the bitstream parser (for instance in the form of a grammar which is interpreted or compiled on the fly), but standard blocks for the remaining decoder modules.

Figure A.1: A general view of an RMC bitstream
A.3 An RMC bitstream

The novelty of RMC is that instead of a decoder being rigidly specified, its architecture is transmitted with the encoded data, to enable reconfiguration on-the-fly. In other words, an RMC bitstream is essentially self-describing, in that its structure and that of the decoder are both transmitted as part of the bitstream (Figure A.1). The decoder structure is written using the Decoder Description Language (DDL) [116], which is an XML dialect, described in Section A.5.1. The structure of the compressed content, on the other hand, is described using BSDL [40], which is discussed in Section A.6. It is this latter description, along with the generation of the Parser block (that is, the elements shaded in green in Figure A.1) that are the specific contributions of this chapter.

In the RMC framework, the receiver device gets the decoder description which fully specifies the architecture of the decoder. In order to instantiate the decoder, the receiver then needs an implementation of the standard library of building blocks specified by RMC. This library is normatively specified using CAL (see Section A.4), which can be directly synthesised into both hardware (VHDL) and software (C, C++, Java, and so on—Figure A.2) by using appropriate tools. Device vendors are, however, free to provide alternative implementations of the standard library that are optimised for their particular platform.

An appropriate level of granularity for blocks within the standard library is important, to enable efficient reuse within the RMC framework. If the library is too coarse,
modules will be too large to allow reuse between different codecs. On the other hand, if the granularity is too fine, the number of modules in the library will be too large for an efficient and practical reconfiguration process, and may obscure the desired high-level description of the RMC decoder.

A.4 The CAL Actor Language

One fundamental component of the RMC framework is the standard library of coding tools that are the *high level* building blocks of a codec. For this library a syntax is required to specify each algorithm and its interfaces, in such a way that algorithms
may be combined easily, yet correctly. Traditional libraries composed of C functions or C++ classes are inadequate, because they require too much integration overhead to yield a *working* codec model. For these reasons, CAL [170] was chosen over C/C++ for specifying the RMC standard library. This section presents the fundamental characteristics of CAL, and the features that make it suitable for RMC.

### A.4.1 Dataflow oriented processing

Looking for high level descriptions of MPEG codecs leads naturally to a *dataflow* processing paradigm. This is not surprising since the fundamental operation of such codecs is to transform a stream of data in the compressed domain to a stream of decoded audio or video (or vice versa). Furthermore, this transformation is characterised by a sequence of operations that are repeated for each unit in the stream.

CAL is a language used to define the behavior of dataflow components called *actors*, which are modular components that encapsulates their own state. That is, an actor can neither read nor modify the state of any other actor. The only interaction between actors is via messages (known in CAL as *tokens*) which are passed from an output of one actor to an input of another. The behavior of an actor is defined in terms of a set of *actions*, at most one of which is active at any point in time. The operations an action can perform are to consume (read) input tokens, modify internal state, produce output tokens, and interact with the underlying platform on which the actor is running. Examples of such interaction include reading the incoming RMC bitstream or rendering decoded output.
After an action completes, the next action to be executed (fired) depends on

- the availability of token(s) at the requisite input(s);
- the value of input tokens;
- the state of the actor; and
- the priority of each action.

An actor may contain any number of actions. Its execution follows a cycle:

(a) determine, for each action, whether it is enabled, by testing all the conditions specified in that action;

(b) execute one enabled action (if any); go to (a).

The selection order and the firing conditions for actions form the core of the design of an actor. CAL provides several constructs for describing action selection, including:

- action guards: conditions on the values of input tokens and/or the values of actor state variables, that need to be true for an action to be enabled;
- a finite state machine, expressed as a set of transitions from one state to another. The condition for each transition is specified by the guards of an action, and when a transition is made, the associated action is fired; and
- action priorities: actions may be related to each other by a partial priority order, such that an action will only execute if no higher-priority action can execute.
In this way, the process of action selection is specified in a declarative manner by the designer. As a result the actor becomes more compact and easier to understand.

A.4.2 Hierarchical modular design

With CAL, a RMC decoder is composed of a network of independent actors, which interact with each other only via token passing. This approach facilitates modularity, where the internal implementation of any actor can be modified without impacting other actors. The behavioral description of an actor, and the architecture of the system are thus completely separate. In contrast, the reference implementations of existing MPEG codecs (written in C or C++) make extensive use of shared memory and are difficult or impossible to componentise.

A.4.3 Communication protocols

Interaction between actors is solely via FIFO channels connecting output ports to input ports. The atomic unit of data sent across these channels is called a *token*, which may be a simple value (such as an integer), an arbitrarily complex data structure, or even a function or procedure (borrowing from the functional programming paradigm).

When a token is produced at an output port, it is delivered to the queue at each input port to which it is connected. The token remains in the queue(s) until it is consumed by the actor that owns that queue.
A.4.4 Nondeterministic scheduling and explicit parallelism

Notwithstanding the firing conditions and schedules discussed above (in A.4.1), the order of execution for actions is nondeterministic. This provides the designer of an RMC decoder great flexibility to schedule action execution according to the particular requirements and constraints of the hardware/software. In terms of the former, this allows better optimisation of area, throughput, power consumption, latency, and so on.

Moreover, a codec specified as a network of CAL actors explicitly exposes parallelism by virtue of the independence of different actors. This parallelism can be exploited if desired, by specific implementations. This is not possible with monolithic C/C++ specifications, where the identification of parallelism is a significant and resource-intensive task.

A.4.5 Summary: CAL

To summarise, CAL is a language that

- is based on dataflow processing primitives;
- facilitates top-down (block diagram) design;
- encapsulates processing tasks in units called actors;
- facilitates parallelisation both in terms of development (i.e. actors may be written in parallel by different authors), and operation (actors may be executed
A.5 The Reconfigurable Media Coding framework

Like previous MPEG coding tools, RMC specifies the operation of the decoder and the bitstream syntax, leaving the particulars of the encoder open to proprietary com-
petitive advantage. However, unlike previous tools, RMC does not itself define a new codec. Instead, RMC provides a framework to allow content providers to define a multitude of different codecs, by combining together blocks (actors) from the standard library.

There are two slightly different models for an RMC decoder (Figure A.3). In the abstract model used for the reference software, decoder actors are instantiated directly from the reference CAL library. The bitstream schema is transformed into a parser actor (see A.7), and the actors run on an interpreter.

On the other hand, device vendors implementing RMC have considerable latitude to optimise the decoder execution environment. Instead of instantiating CAL blocks, the standard library is implemented in a format native to the environment. The library may be synthesised from the reference library (for example, a CAL to VHDL compiler is available [171]), and/or further optimised as part of the decoder implementation. The interface of each actor in the standard library (in terms of inputs, outputs and behavior) is normatively defined, but the implementation is not. Equally, the bitstream schema bitstream metalanguage and semantics are normative, but the parsing process is fully implementation dependent.

A.5.1 Decoder Description Language

The second fundamental component of the RMC framework is the language used for the description of the decoder as a network of coding tools (i.e. actors). This Decoder Description Language (DDL) specified in RMC is an XML dialect that describes an
interconnected network of standard library components, which together represent a complete decoder. A DDL description of the intended decoder configuration is transmitted as part of an RMC bitstream, and is used by the decoder to instantiate and interconnect the appropriate modules from the standard library. DDL can also be used recursively; that is, an Actor may be defined as a composition of other actors, with the interconnections specified by DDL. In this case, the DDL itself declares input and output ports.

DDL provides a facility for declaring parameters, and passing parameters to actors in the network. This is useful for declaring values that are constant for a particular instantiation of an actor, but may vary between different instantiations. For example, a vendor may have a number of different RMC-enabled devices, with varying screen resolution or audio depth. In this case the vendor may implement certain actors in the standard library only once, but with parameters to fix the varying quantities. Parameter values are denoted by expressions, which may depend on the values of other parameters and global or local variables.

### A.6 Bitstream Syntax Description

The final part of a RMC bitstream is a description of the syntax used for the content data. This information allows a RMC decoder to parse the bitstream into fields, as well as group individual fields into semantic units. There are numerous syntax description languages available; see Section 2.1 for an extensive discussion, which
concludes that BSDL has a number of features that make it more desirable in the context of this thesis.

This section presents an assessment of the suitability of BSDL for syntax description specifically within the RMC framework. There are fundamentally two questions that must be addressed here:

1. Is BSDL capable of describing real-world multimedia formats in complete detail? and
2. If so, can the resulting BSDL Schema be used efficiently by a generic parser module to read bitstreams? That is, does BSDL allow efficient RMC implementations?

There are also other secondary considerations such as readability (i.e. complexity), ease of development and debugging, and verbosity.

The MPEG-4 Video Simple Profile [167] standard is used as a test subject to address the questions raised above. Although RMC is designed for the creation of new codecs, it is prudent to first establish that it may be used successfully to describe existing ones. The MPEG-4 standard specifies syntax predominantly via tables of pseudocode, an example of which is shown in Table A.1, although some parts are described in prose (such as DCT coefficient decoding), and others using look-up tables (typically VLCs). The pseudocode shown in the figure corresponds to the BSDL Schema extract shown in Listing A.1. BSDL uses declarative structures (sequence, choice) rather than imperative constructs (for, do..while), but it is straightforward to
develop a BSDL Schema based on the pseudocode.

BSDL is also able to express the DCT decoding process specified in prose in the MPEG-4 standard. In doing so, it in fact provides a significantly more objective specification mechanism than the standard itself. BSDL does not directly support VLC decoding. However, it provides an extension mechanism for such cases to allow the parsing process to be specified in an external language (in this case CAL). Using this, it is simple to express the MPEG-4 VLC tables in the BSDL Schema in a form that an RMC decoder is able to process.

A complete BSDL Schema for MPEG-4 Video Simple Profile has been developed and tested, verifying that BSDL is sufficiently descriptive to be used with real-world media, and it may be found in Appendix J. The use of BSDL in an RMC parser module is described in the following section (A.7). Other work [172] extends this
validation to H.264/AVC, and demonstrates the extensibility of the RMC framework by deploying content encoded using 4:2:2 and 4:4:4 chroma subsampling: features that don’t exist within the standardised version of MPEG-4. In an RMC bitstream, this means adding extra chroma blocks in each Macroblock, and changing the chroma block pattern header field. In the BSDL Schema, this simply requires changing the \texttt{maxOccurs} value on the chroma blocks.

Informally, this Listing \texttt{A.1} states that a Video Object Layer is made up of either a long header or a short header, as well as many Video Object Plane structures (VOP is MPEG-4 terminology for a video frame). The choice between a long or short header is made on the basis of whether the subsequent bits in the bitstream are equal to the value \texttt{0x00000120} (this is in fact the start code that is subsequently stored as the first field of a long header). The variable \texttt{($mbCount$)} is computed on the basis of prior fields in the long header, and is used when parsing VOPs to determine the number of Macroblocks (MBs) to parse.
Listing A.1: Part of the BSDL Schema for MPEG-4 Video

<complexType name="VideoObjectPlaneType">
  <sequence>
    <element name="vopHdr" type="VOPHeaderType"/>
    <element name="motionShapeTexture" minOccurs="0" type="MotionShapeTextureType" bs2:if="$vopCoded=1"/>
  </sequence>
</complexType>

<complexType name="VOPHeaderType">
  <sequence>
    <element name="vopSC" type=" SCType"/>
    <element name="vopCoded" type="bs1:b1" bs0:variable="true"/>
  </sequence>
</complexType>

<complexType name="MotionShapeTextureType">
  <sequence>
    <element name="MB" type="MBType" rmc:port="MB"
      maxOccurs="unbounded" bs2:nOccurs="$mbCount"/>
  </sequence>
</complexType>

<complexType name="MBType">
  <sequence>
    <element name="horizMVData" type="MVDataType"/>
  </sequence>
</complexType>

<simpleType name="MVDataType">
  <restriction>
    <annotation>
      <appinfo>
        <bs1:script ref="mvData.cal" lang="cal"/>
      </appinfo>
    </annotation>
  </restriction>
</simpleType>
A.7 Parser Generation

The BSDL Schema transmitted with an RMC bitstream contains all of the information necessary to parse the rest of the bitstream. The decoder translates this schema into a parser block whose task is to convert the raw bits into structured data that may be processed by subsequent decoder modules. Although this translation is relatively involved (Figure A.4), the declarative model shared by both BSDL and CAL means that the translation process may be efficiently specified. Each component is implemented in a separate XSLT stylesheet, which is then imported by a master sheet that coordinates the overall process.

The following discussion provides examples of each component. In each example, the input is shaded in green, the stylesheet in blue, and the output in yellow. The complete schema and stylesheets are too voluminous to be included in this chapter, but are provided in Appendix J.
A.7.1 Preprocessing

Preprocessing is the first operation conducted by the stylesheet. In general, a BSDL Schema may be composed of a number of separate schemata, which are imported by a master document (much the same as the stylesheet). The preprocessing stage is therefore necessary to collect the individual schemata into a single intermediate tree, taking care to correctly manage the namespace of each component Schema. The preprocessor also performs a number of other tasks, including assigning names to anonymous types and structures (so that they may be referred to by the FSM transition set), resolving inheritance relationships, and removing structures which are not significant to the parsing process.

Listing A.2(a): Input schema fragment showing an include directive

```xml
<xsd:schema targetNamespace="urn:mpeg:mpeg21:2007:01−RMC−MPEG4Visual">
  <xsd:include schemaLocation="mpeg−4_visual_simple_types.xsd"/>
</xsd:schema>
```

Listing A.2(b): XSLT fragment showing preprocessor templates

```xml
<xsl:stylesheet>
  <xsl:template match="xsd:include" mode="#all"/>
  <xsl:variable name="targetNamespace" select="/xsd:schema/@targetNamespace"/>
  <xsl:variable name="includedDoc" select="document(@schemaLocation)/xsd:schema/*"/>
  <xsl:apply-templates select="$includedDoc" mode="#current"/>
  <xsl:with-param name="prefix" select="rmc:findPrefix($targetNamespace,.,$includedDoc)" tunnel="yes"/>
  <xsl:with-param name="namespace" select="$targetNamespace" tunnel="yes"/>
</xsl:apply-templates>
```
Listing A.2(c): Fragment of intermediate result tree composed of base and included schema components

A.7.2 Finite State Machine

Finite State Machine (FSM) design is the major component of the parser actor. The FSM schedules the reading of bits from the input bitstream into the fields in the various output structures, along with all other components of the actor. The FSM is specified as a set of transitions, where each transition has an initial state, a final state, and an action. Computing the FSM from a BSDL Schema has several components, each of which are highlighted in bold within Figure A.4.
A.7.2.1 Linearisation

The state pattern of the FSM is predominantly linear: the first field is read, then the next field, then the next, and so on. Consequently, the hierarchical structure of the BSDL Schema must be converted into a linear sequence of read instructions, from which the FSM Transition and Action sets may be built. This operation is performed by the Linearise component, which reads the preprocessed BSDL Schema and outputs an intermediate data structure comprising the linear sequence of read actions. It is from this intermediate structure that the FSM is assembled.

Listing A.3(a): Input schema fragment showing a complex type and sequence of particles

```xml
<xsd:complexType name="VideoObjectPlaneType">
  <xsd:sequence>
    <xsd:element name="vopHeader" type="m4v:VideoObjectPlaneHeaderType" rmc:port="vop"/>
    <xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coded=1">
      <xsd:element name="motion_shape_texture" type="MotionShapeTextureType"/>
      <!--...-->
    </xsd:sequence>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded" bs2:ifNext="7F">
      <xsd:element name="stuffing_bits" type="bs1:align8"/>
      <xsd:element name="stuffing_byte" type="m4v:hex1" fixed="7F" minOccurs="0" maxOccurs="unbounded" bs2:ifNext="7F"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
```

Listing A.3(b): XSLT fragment for the linearise component

```xml
<!-- create a read element for the particle -->
<xsl:template match="xsd:element | xsd:group | xsd:sequence | xsd:all" priority="5" mode="linearise">
  <xsl:param name="stack" select="()"/>
  <xsl:param name="depth" select="0"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
</xsl:template>
```
<read>
  <xsl:attribute name="name" select="rmc:itemName($newStack)"/>
  <xsl:attribute name="element" select="local-name()='element'"/>
  <xsl:attribute name="depth" select="$depth"/>
  <xsl:apply-templates select="@* | *" mode="annotateRead"/>
</read>

<!-- Find the next element to read — varies according to type -->
<xsl:next-match>
  <xsl:with-param name="stack" select="$newStack"/>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:next-match>
</xsl:template>

<!-- annotate the read element with an attribute if a test is required before the element is read -->
<xsl:template match="@minOccurs" mode="annotateRead">
  <xsl:if test=".=0">
    <xsl:attribute name="testRequired">true</xsl:attribute>
    <xsl:copy/>
  </xsl:if>
</xsl:template>

Listing A.3(c): Fragment of intermediate result tree composed of read instructions

<read name="vop" element="true" depth="0" type="complex"
  typeName="m4v:VideoObjectPlaneType"/>
<read name="vop.vopHeader" element="true" depth="1"></read>
<read name="vop.vopHeader" depth="1" rmc:port="vop"></read> <!--[a]-->
<read name="vop.sequence1" element="false" depth="1"
  testRequired="true" minOccurs="0"
  bs2:if="$m4v:vop.coded=1"/>
<read name="vop.sequence1.motion_shape_texture" element="true"
  depth="1"></read>
<read name="vop.stuffing_bits" element="true" depth="1"></read>
<read name="vop.stuffing_byte" element="true" depth="1"
  testRequired="true"
  minOccurs="0" multiple="true" maxOccurs="4.2E9" bs2:ifNext="7F"/>
<read name="vop.stuffing_byte" again="true" testRequired="true"></read> <!--[b]-->

The first template in Listing A.3(b) creates the read elements and assigns name, element and depth attributes, before passing control to any applicable templates
Tearing it all down again

with the annotateRead mode. One such template is shown here, which creates a testRequired="true" attribute if the input element is optional and so the parser must test whether it exists. There are numerous other linearise templates that are not shown in Listing A.3(b) because of space limitations. These are responsible for other elements and attributes in parts of the schema in Listing A.1, for example

A.3(c)--[a] a template to create a read element which is responsible for outputting the completed vop.vopHeader to an output port; and

A.3(c)--[b] a template to create a read element that tests whether to read vop.stuffing_byte again.

Listing A.4(a): Intermediate result tree fragment composed of read instructions

<!-- same as Listing A.3(c) -->

Listing A.4(b): XSLT fragment showing FSM transition templates

```xml
<!-- transition from this element to the next read element -->
<xsl:template match="read" mode="fsm">
  <xsl:call-template name="fsmTransition">
    <xsl:with-param name="current" select="."/>
    <xsl:with-param name="next" select="rmc:nextReadElement(.)"/>
  </xsl:call-template>
</xsl:template>

<!-- transition from this to this.exists, and a skip transition from this to next -->
<xsl:template match="read[@testRequired='true']" priority="10" mode="fsm">
  <xsl:variable name="next" select="rmc:nextReadElement(.)"/>
  <xsl:variable name="nextNotReadAgain" select="if ($next/@again='true') then rmc:nextReadElement($next) else $next"/>
  <xsl:call-template name="fsmTransition">
    <xsl:with-param name="current" select="."/>
  </xsl:call-template>
</xsl:template>
```
Listing A.4(c): Fragment of CAL output showing FSM schedule

```
schedule fsm vop_exists :
  vop_exists (vop.read) --> vop_vopHeader_exists
  vop_vopHeader_exists (vop.vopHeader.read) --> vop_vopHeader_output
  vop_vopHeader_output (vop.vopHeader.output) --> vop_sequence1
  vop_sequence1 (vop.sequence1.skip) -->
    vop_sequence1_motion_shape_texture_exists
  vop_sequence1 (vop.sequence1.test) --> vop_sequence1_exists
  vop_sequence1_exists (vop.sequence1.read) -->
    vop_sequence1_motion_shape_texture_exists
    (vop.sequence1.motion_shape_texture.read) -->
    vop_stuffing_bits_exists
  vop_stuffing_bits_exists (vop.stuffing_bits.read) --> vop_stuffing_byte
  vop_stuffing_byte (vop.stuffing_byte.skip) --> exit
  vop_stuffing_byte (vop.stuffing_byte.test) --> vop_stuffing_byte_exists
  vop_stuffing_byte_exists (vop.stuffing_byte.read) --> vop_stuffing_byte
end
```

A.7.2.2 Guards and Behaviour

Actions scheduled by the FSM control the next-state decision mechanism via their Guard expressions, which are built from the control-flow constructs in the BSDL.
Tearing it all down again

Schema (if, ifNext, nOccurs and length). The Behaviour of each action is to complete such tasks as storing data in the appropriate location in the output structure and/or variables, and setting the number of bits to be read for the subsequent field.

**Listing A.5(a)**: Intermediate result tree fragment composed of read instructions

```
<!-- same as Listing A.3(c) -->
```

**Listing A.5(b)**: XSLT fragment showing action templates

```xml
<!-- Optional elements need a skip action, a test action, and a read action -->
<xsl:template match="read[@testRequired='true']" priority="10"
  mode="buildActions">
  <!-- Skip action: set the count for the following::read[1] element -->
  <xsl:call-template name="action">
    <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
    <xsl:with-param name="guard">
      <xsl:apply-templates select="." mode="skipGuard"/>
    </xsl:with-param>
    <xsl:call-template name="setCountForNextRead"/>
  </xsl:call-template>
</xsl:template>

<!-- Test action: set the count for this element -->
<xsl:call-template name="action">
  <xsl:with-param name="name" select="concat(@name,'&testActionSuffix;')"/>
  <xsl:with-param name="guard">
    <xsl:apply-templates select="." mode="testGuard"/>
  </xsl:with-param>
  <xsl:call-template name="setCountForNextRead"/>
</xsl:call-template>

<!-- read action -->
<xsl:next-match/>
```

```
<!-- Default read action -->
<xsl:template match="read" mode="buildActions">
  <xsl:call-template name="action"/>
```
Listing A.5(c): Output CAL fragment showing several actions

vop.read: action =>
do
  current[0] := VideoObjectPlaneType();
end

/* ... */

vop.stuffing_byte.test: action =>
guard
  readAheadDone() and
  VOP_STUFFING_BYTE_IFNEXT_MIN_VALUE <= readAheadResult()
  and
  VOP_STUFFING_BYTE_IFNEXT_MAX_VALUE >= readAheadResult()
do
  setRead(XSD_BYTE_LENGTH);
end

vop.stuffing_byte.skip: action =>
guard
  readAheadDone()
end

vop.stuffing_byte.read: action =>
do
  current[0].stuffing_byte := readResult();
end
Figure A.5: FSM fragment for a choice particle

There are two exceptions to the linear state flow: Choice particles and Union types. Choice particles (such as that in Listing A.1) cause the FSM to diverge from its linear path to one of a number of parallel paths, each of which parse a single option of the choice. For example, Figure A.5 depicts the FSM fragment for the VOL header choice of Listing A.1. Each of the parallel paths has a test action that determines which of the options is selected. As before, the guards on each test action are built from control-flow constructs in the BSDL Schema. BSDL specifies that the order of options within a choice establishes their priority: the first option has priority over the second, and so on. These priorities are recorded in the actor as priorities between the test actions.

Union types are very similar to Choice particles, except that instead of choosing between a number of different objects to instantiate, a Union chooses between a number of different types that a single object could take. For example, a single field could be either 16 bits or 32 bits, depending on the resolution it is required to record. Union types have the same state structure as choice particles, but differ in the composition of their test guards.
A.7.3 Field bit-length

Field bit-length in BSDL is specified indirectly via the `xsd:maxExclusive` facet of XML Schema. A stylesheet component is therefore required to compute the bit-length of simple types within the schema from their `maxExclusive` value. Once computed, the value is stored in a constant identified by the type name, and subsequently used whenever a field of that type is read from the bitstream.

Listing A.6(a): Input schema fragment showing an element with simple type

```xml
<xsl:element name="stuffing_byte" type="xsd:byte" minOccurs="0" maxOccurs="unbounded" bs2:ifNext="7F"/>
```

Listing A.6(b): XSLT fragment showing length constant templates

```xml
<xsl:template name="lengthConstants">
  <xsl:variable name="context" select="."/>
  <!-- create LENGTH constants for simple types -->
  <xsl:apply-templates select="/xsd:schema/xsd:simpleType" mode="length"/>
  <!-- create LENGTH constants for referenced XSD types -->
  <xsl:for-each select="distinct-values(../xs:element/@type)">
    <xsl:variable name="typeQName" select="resolve-QName(.,$context)"/>
    <xsl:if test="namespace-uri-from-QName($typeQName)=$xsdNS;">
      <xsl:call-template name="statement">
        <xsl:with-param name="expressions">
          <xsl:call-template name="variableDeclaration">
            <xsl:with-param name="name" select="concat(rmc:constant(.),&lengthSuffix;)"/>
            <xsl:with-param name="type" select="int"/>
            <xsl:with-param name="initialValue">
              <xsl:call-template name="xsdTypeLength">
                <xsl:with-param name="typeQName" select="$typeQName"/>  
              </xsl:call-template>
            </xsl:with-param>
          </xsl:call-template>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```
<!—–convert the hex value(s) of ifNext to decimal, create constants for the length to read ahead, and the minimum and maximum values to test. —–>
<xsl:template match="*[@bs2:ifNext]" mode="globals">
  <xsl:variable name="ifNext" select="rmc:split(@bs2:ifNext,' ')"/>
  <xsl:call-template name="statement">
    <xsl:with-param name="expressions">
      <xsl:call-template name="variableDeclaration">
        <xsl:with-param name="name" select="concat(rmc:constant(@name), '&ifNextLength;')"/>
        <xsl:with-param name="initialValue" select="rmc:hexByteLength($ifNext[1])"/>
        <xsl:with-param name="type">int</xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
  <xsl:call-template name="statement">
    <xsl:with-param name="expressions">
      <xsl:call-template name="variableDeclaration">
        <xsl:with-param name="name" select="concat(rmc:constant(@name), '&minValue;')"/>
        <xsl:with-param name="initialValue" select="rmc:hexToDec($ifNext[1])"/>
        <xsl:with-param name="type">int</xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
  <xsl:call-template name="statement">
    <xsl:with-param name="expressions">
      <xsl:call-template name="variableDeclaration">
        <xsl:with-param name="name" select="concat(rmc:constant(@name), '&maxValue;')"/>
        <xsl:with-param name="initialValue" select="rmc:hexToDec($ifNext[last()])"/>
        <xsl:with-param name="type">int</xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>
Listing A.6(c): Output CAL fragment showing length constants

```plaintext
int XSD_BYTE_LENGTH := 8;
int VOP_STUFFING_BYTE_IFNEXT_MIN_VALUE := 127;
int VOP_STUFFING_BYTE_IFNEXT_MAX_VALUE := 127;
```

### A.7.4 CAL Templates

Finally, the CAL component declares templates for each of the constructs in the language, such as an FSM schedule, a function call, or an assignment. These templates are called by other components of the stylesheet when building the actor. Collecting all of the CAL syntax into a single stylesheet also means that an alternative stylesheet could be provided in place of the CAL sheet, for example containing templates that output CALML, or potentially even other languages.

Listing A.7(a): Intermediate result tree

```xml
<!−− same as Listing A.3(c) −−>
```

Listing A.7(b): XSLT fragment showing CAL templates for an FSM & transition

```
<xsl:template name="fsm">
  <xsl:param name="initialState"/>
  <xsl:param name="transitions"/>
  <xsl:text>&lt;nl;&amp;tab;schedule fsm &lt;/xsl:text>
  <xsl:value-of select="$initialState"/>
  <xsl:text&gt;&lt;/xsl:text>
  <xsl:value-of select="string−join($transitions,'&amp;nl;')"/>
  <xsl:text>&amp;tab;end&amp;nl;&lt;/xsl:text>
</xsl:template>

<xsl:template name="transition">
  <xsl:param name="from" required="yes"/>
  <xsl:param name="to" required="yes"/>
```

A.8 Conclusion

This chapter assessed the suitability of BSDL for syntax description within the RMC framework, and developed a method for implementing an RMC parser using a BSDL Schema. That is, it addressed the questions of 

a) can BSDL describe complete real-world media formats? and 

b) if so, can the resulting BSDL Schema be used efficiently by a generic parser module to read bitstreams in an RMC implementation?

As seen in Appendix J, BSDL is able to describe the complete detail of the MPEG-4 Video Simple Profile, and this schema has been validated by using it with the BinToBSD [40] parser against real content.

Furthermore, Section A.7 developed a methodology for transforming a BSDL schema into a CAL Actor, which implements a parser for the specified bitstream structure.
The approach uses XSLT to transform the BSDL into CAL or CALML. It is based around a Finite State Machine where each state either parses a field from the bit-stream, or outputs completed structures to an output port. These structures are hierarchical and object-based and are passed to downstream actors in the RMC decoder instance.

### A.8.1 Future Work

The ultimate goal for RMC is to realise a fully programmable decoder specification model as outlined in Section A.2. Doing so would substantially shorten the workflow from multimedia research to consumer, by obviating the need for a lengthy standardisation process in order to ensure interoperability for new technology (Figure A.6a). Instead, new multimedia technology could be immediately deployed using RMC tools (Figure A.6b). Realising this vision requires further work, particularly in on-the-fly reconfigurability for FPGA-based systems, to allow custom actors to
be instantiated directly from code within the RMC bitstream. However, the well defined component model of the CAL language, and the reconfigurability of an RMC bitstream described by BSDL are significant steps toward this goal.

Furthermore, RMC has thus far focused on video coding. In order to complete a Reconfigurable Media Coding framework, RMC must be extended to allow the reconfigurable description of audio codecs. As seen in the other chapters of this thesis, BSDL is widely used with audio formats at a high level; validating BSDL for use with audio in RMC will require its use to describe a complex audio format in its entirety. The CAL reference library must also be updated with modules to implement the various functions of audio decoders.

Finally, RMC suggests abundant possibilities for research on how to exploit reconfigurability to achieve greater coding efficiency, robustness and/or flexibility. Given an RMC decoding device, the way that content is encoded can be greatly altered to better suit the capabilities of the device or network, or the particular characteristics of the content.
Appendix B

Data tables for BSDL
Table B.1: Data table for Figure 3.4

<table>
<thead>
<tr>
<th>Samples</th>
<th>Duration (s)</th>
<th>base</th>
<th>3.3.1 userType</th>
<th>3.3.2 variables</th>
<th>3.3.3 bitLength</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>1.69</td>
<td>1.64</td>
<td>1.66</td>
<td>0.62</td>
<td>0.68</td>
<td>0.45</td>
</tr>
<tr>
<td>101</td>
<td>3.35</td>
<td>4.63</td>
<td>4.61</td>
<td>1.06</td>
<td>1.38</td>
<td>0.77</td>
</tr>
<tr>
<td>151</td>
<td>5.01</td>
<td>9.72</td>
<td>9.92</td>
<td>1.43</td>
<td>2.79</td>
<td>1.03</td>
</tr>
<tr>
<td>201</td>
<td>6.67</td>
<td>14.24</td>
<td>15.68</td>
<td>1.89</td>
<td>4.29</td>
<td>1.29</td>
</tr>
<tr>
<td>251</td>
<td>8.33</td>
<td>16.89</td>
<td>13.88</td>
<td>1.34</td>
<td>3.86</td>
<td>1.27</td>
</tr>
<tr>
<td>301</td>
<td>9.99</td>
<td>21.21</td>
<td>21.58</td>
<td>1.61</td>
<td>5.69</td>
<td>1.96</td>
</tr>
<tr>
<td>401</td>
<td>13.31</td>
<td>68.29</td>
<td>49.53</td>
<td>2.58</td>
<td>10.10</td>
<td>3.04</td>
</tr>
<tr>
<td>501</td>
<td>16.63</td>
<td>109.13</td>
<td>82.93</td>
<td>5.54</td>
<td>27.27</td>
<td>2.49</td>
</tr>
<tr>
<td>601</td>
<td>19.95</td>
<td>93.68</td>
<td>95.43</td>
<td>3.54</td>
<td>23.36</td>
<td>3.43</td>
</tr>
<tr>
<td>701</td>
<td>23.28</td>
<td>130.85</td>
<td>133.53</td>
<td>4.09</td>
<td>32.84</td>
<td>3.93</td>
</tr>
<tr>
<td>751</td>
<td>24.94</td>
<td>153.26</td>
<td>156.43</td>
<td>3.79</td>
<td>38.12</td>
<td>4.58</td>
</tr>
<tr>
<td>801</td>
<td>26.60</td>
<td>175.02</td>
<td>179.15</td>
<td>4.59</td>
<td>43.46</td>
<td>4.35</td>
</tr>
<tr>
<td>901</td>
<td>29.92</td>
<td>230.34</td>
<td>229.96</td>
<td>5.06</td>
<td>57.06</td>
<td>4.42</td>
</tr>
<tr>
<td>1001</td>
<td>33.24</td>
<td>370.49</td>
<td>463.41</td>
<td>9.43</td>
<td>112.59</td>
<td>4.16</td>
</tr>
<tr>
<td>1101</td>
<td>36.56</td>
<td>326.10</td>
<td>335.93</td>
<td>5.62</td>
<td>80.59</td>
<td>5.46</td>
</tr>
<tr>
<td>1201</td>
<td>39.88</td>
<td>394.78</td>
<td>399.25</td>
<td>6.16</td>
<td>96.84</td>
<td>5.98</td>
</tr>
<tr>
<td>1251</td>
<td>41.54</td>
<td>427.65</td>
<td>442.97</td>
<td>6.33</td>
<td>104.77</td>
<td>6.68</td>
</tr>
<tr>
<td>1301</td>
<td>43.20</td>
<td>464.01</td>
<td>473.83</td>
<td>6.61</td>
<td>114.60</td>
<td>6.29</td>
</tr>
<tr>
<td>1501</td>
<td>49.84</td>
<td>992.89</td>
<td>943.38</td>
<td>7.51</td>
<td>153.54</td>
<td>6.46</td>
</tr>
<tr>
<td>1751</td>
<td>58.14</td>
<td>877.24</td>
<td>929.59</td>
<td>8.64</td>
<td>212.89</td>
<td>9.01</td>
</tr>
<tr>
<td>1951</td>
<td>64.78</td>
<td>1095.88</td>
<td>1119.50</td>
<td>9.39</td>
<td>270.45</td>
<td>7.42</td>
</tr>
<tr>
<td>2151</td>
<td>71.42</td>
<td>1378.18</td>
<td>1368.98</td>
<td>10.33</td>
<td>322.38</td>
<td>10.87</td>
</tr>
<tr>
<td>2451</td>
<td>81.38</td>
<td>1793.92</td>
<td>1795.89</td>
<td>11.89</td>
<td>418.99</td>
<td>8.14</td>
</tr>
<tr>
<td>2651</td>
<td>88.02</td>
<td>2110.09</td>
<td>2116.53</td>
<td>12.45</td>
<td>503.17</td>
<td>13.24</td>
</tr>
<tr>
<td>2951</td>
<td>97.98</td>
<td>2562.47</td>
<td>2607.10</td>
<td>13.67</td>
<td>614.78</td>
<td>9.87</td>
</tr>
<tr>
<td>3151</td>
<td>104.62</td>
<td>2870.45</td>
<td>2897.29</td>
<td>14.74</td>
<td>703.94</td>
<td>14.13</td>
</tr>
<tr>
<td>3451</td>
<td>114.58</td>
<td>3611.72</td>
<td>3614.54</td>
<td>15.82</td>
<td>833.76</td>
<td>16.99</td>
</tr>
<tr>
<td>3651</td>
<td>121.22</td>
<td>3983.59</td>
<td>4055.82</td>
<td>16.51</td>
<td>941.51</td>
<td>16.13</td>
</tr>
<tr>
<td>3951</td>
<td>131.18</td>
<td>4679.59</td>
<td>4703.13</td>
<td>17.97</td>
<td>1101.91</td>
<td>16.39</td>
</tr>
</tbody>
</table>

1.3% overhead, O(n) complexity, 75% reduction
### Table B.2: Data table for Figure 3.5

<table>
<thead>
<tr>
<th>Samples</th>
<th>Duration (h:m:s)</th>
<th>Avg Parse Time (s)</th>
<th>Avg Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>all</td>
<td>lowMemory</td>
</tr>
<tr>
<td>100</td>
<td>00:00:03</td>
<td>0.70</td>
<td>0.53</td>
</tr>
<tr>
<td>500</td>
<td>00:00:17</td>
<td>2.26</td>
<td>2.35</td>
</tr>
<tr>
<td>1000</td>
<td>00:00:33</td>
<td>4.75</td>
<td>4.68</td>
</tr>
<tr>
<td>1950</td>
<td>00:01:05</td>
<td>8.69</td>
<td>8.74</td>
</tr>
<tr>
<td>2950</td>
<td>00:01:38</td>
<td>12.85</td>
<td>13.26</td>
</tr>
<tr>
<td>3950</td>
<td>00:02:11</td>
<td>17.01</td>
<td>17.65</td>
</tr>
<tr>
<td>15800</td>
<td>00:08:45</td>
<td>65.47</td>
<td>69.06</td>
</tr>
<tr>
<td>31600</td>
<td>00:17:29</td>
<td>129.87</td>
<td>107.87</td>
</tr>
<tr>
<td>63203</td>
<td>00:34:59</td>
<td>178.17</td>
<td>240.04</td>
</tr>
<tr>
<td>126403</td>
<td>01:09:57</td>
<td>387.35</td>
<td>440.67</td>
</tr>
</tbody>
</table>
Appendix C

Bitstream Binding Language Specification
C.1 Terms and definitions ........................................................................................................ C-4
C.2 Conventions ....................................................................................................................... C-5
C.2.1 Documentation conventions ........................................................................................ C-5
C.2.2 Namespace prefix conventions ................................................................................... C-7
C.3 Bitstream Binding Language ............................................................................................. C-8
C.3.1 Introduction .................................................................................................................. C-8
C.3.2 Overview ..................................................................................................................... C-8
C.3.3 BBL .............................................................................................................................. C-8
C.3.4 BBL Documents .......................................................................................................... C-10
C.3.5 Processing of Binary Resources ................................................................................ C-10
C.3.6 BBL Processing Order ............................................................................................... C-11
C.3.7 Bitstream Binding Language Definition ....................................................................... C-11
C.3.7.1 Introduction ............................................................................................................. C-11
C.3.7.2 Validation ............................................................................................................... C-12
C.3.7.3 Document Modularity ............................................................................................ C-12
C.3.7.4 Use of “?” to delimit XPath expressions within attribute values ......................... C-12
C.3.7.5 Use of variables and functions within XPath expressions ...................................... C-12
C.3.7.6 xmlSource and binarySource Attributes ............................................................... C-14
C.3.7.7 sourceElt Type ....................................................................................................... C-16
C.3.7.8 bbl Element .......................................................................................................... C-17
C.3.7.9 instance Element ................................................................................................. C-17
C.3.7.10 binding Element ................................................................................................. C-19
C.3.7.11 variables Element ............................................................................................... C-20
C.3.7.12 define Element .................................................................................................... C-21
C.3.7.13 assign Element .................................................................................................... C-22
C.3.7.14 declarations Element ......................................................................................... C-23
C.3.7.15 register Element ................................................................................................ C-23
C.3.7.16 handler Element ............................................................................................... C-24
C.3.7.17 bsd Element ....................................................................................................... C-25
C.3.7.18 packet Element ..................................................................................................... C-26
C.3.7.19 packetStream Element ...................................................................................... C-28
C.3.7.20 handlerParams Element ................................................................................... C-30
C.3.7.21 content Element ................................................................................................ C-30
C.3.7.22 contentTemplate Element ................................................................................. C-31
C.3.7.23 include Element ................................................................................................ C-32
C.3.7.24 timing Element .................................................................................................... C-35
C.3.7.25 delTimes Element .............................................................................................. C-35
C.3.7.26 durations Element ............................................................................................. C-36
C.3.7.27 fragmentation Element ...................................................................................... C-37
C.3.7.28 size Element ........................................................................................................ C-37
C.3.7.29  duration Element ......................................................................................................................C-37
C.3.7.30  count Element ......................................................................................................................C-37
C.3.7.31  constraint Element ...............................................................................................................C-39
C.3.7.32  bind Element ........................................................................................................................C-40
C.3.7.33  value-of Element ...................................................................................................................C-41
C.3.7.34  attribute Element .................................................................................................................C-42
C.3.7.35  BBL provided Attributes .........................................................................................................C-43
C.4  Definition of handlers for the Bitstream Binding Language ..........................................................C-44
C.4.1  Introduction .....................................................................................................................................C-44
C.4.2  MPEG-2 Transport Stream handler .............................................................................................C-44
C.4.3  Introduction .....................................................................................................................................C-44
C.4.4  Handler type Declaration ............................................................................................................C-44
C.4.5  Handler Parameter Syntax ..........................................................................................................C-45
C.4.6  Handler Parameter Semantics .....................................................................................................C-46
C.4.7  Real Time Protocol handler ........................................................................................................C-48
C.4.8  Introduction .....................................................................................................................................C-48
C.4.9  Handler type Declaration ............................................................................................................C-48
C.4.10 Handler Parameter Syntax ..........................................................................................................C-49
C.4.11 Handler Parameter Semantics .....................................................................................................C-49

Bibliography ..................................................................................................................................................C-52
C.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

**Binding**
a set of abstract Bitstream Binding Language processing instructions which map content of a particular arrangement (e.g., a particular content format, or a set of XML documents with similar structure) into a collection of packets to be output to one or more handlers.

**Binary Document**
a binary resource which can be processed using the Bitstream Binding Language via a BS Schema to describe the syntactical structure of the resource.

**Delivery Time**
the point in time when a Packet becomes available to an MPEG-21 Peer for consumption.

**Document**
either an XML Document, as defined by W3C XML, or a Binary Document.

**Document Fragment**
an XML Document Fragment, as defined by W3C DOM.

**Duration**
the length of time for which an object is active.

**NOTE** In BBL, individual Elements may be assigned Duration, which aggregate to provide the duration of a packet. If an Element has no explicitly associated Duration, it is assumed to have a Duration of zero.

**Element**
an XML Element, as defined by W3C XML.

**Fragmentation**
authoring process by which a Digital Item is split into Packets meaningful for consumption purposes.

**NOTE** This process can attach time information to the output Packets indicating the point in time when they become available to the MPEG-21 Peer for consumption.

**Fragmentation Constraint**
a rule which constrains the way in which content is fragmented.

**EXAMPLE** One possible constraint is that a particular element and its subtree should be “unbroken” — i.e., transmitted in a single packet without fragmentation.

**Fragmentation Rule**
a criteria which defines conditions that a content fragment must fulfil. If the rule is not fulfilled, the content must be further fragmented.
EXAMPLE An example of this is a maximum size (in bytes) for any fragment.

**Handler**

A plug-in object to a BBL engine which receives a byte array representing the content of a Packet, along with the Delivery Time and Repetition Period for the Packet, and outputs the Packet to a transport stream.

**NOTE** This process can be guided by parameters provided either at a global or local level.

**Instance**

BBL Document which processes one or more concrete source documents to form a collection of packets which are output to one or more Handlers.

**NOTE** If required, BBL Instances refer to BBL Bindings to facilitate this processing.

**Map**

Process of fragmenting a Source Document, assigning temporal and other parameters to fragments, and inserting the fragments into a specific location of an outgoing stream.

**Packet**

A fragment of a Digital Item (DID, metadata and/or resources), which is delivered to an MPEG-21 Peer, and to which temporal information (e.g., Delivery Time Stamp) may be attached.

**Packet Stream**

A sequence of Packets which possess time-invariant properties such that BBL is able to operate upon the sequence as a whole.

**Processable**

A Packet is Processable if it does not rely upon Packets which have not yet been received.

**Source Document**

Either an XML Document or a Binary Document against which W3C XPATH expressions in BBL attributes are evaluated.

**NOTE** In the case of a Binary Document, W3C XPATH expressions are evaluated via a BS Schema which has been associated with the prefix(es) used in the W3C XPATH expression.

### C.2 Conventions

#### C.2.1 Documentation conventions

The syntax of each element in the Bitstream Binding Language is specified using the constructs provided by W3C XMLSCHEMA.

Element names and attribute names in the representation are in this typeface.

The syntax of each element in the Bitstream Binding Language is specified using the following format.
### Table 1 — Example element specification

<table>
<thead>
<tr>
<th>Children</th>
<th>&lt;CHILD1&gt; &lt;CHILD2&gt; &lt;CHILD3&gt; &lt;CHILD4&gt; &lt;CHILD5&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by</td>
<td>&lt;GRANDPARENT1&gt; &lt;GRANDPARENT2&gt;</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
</tbody>
</table>

#### Source

```xml
<xs:element name="PARENT">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="CHILD1" minOccurs="0"/>
      <xs:element ref="CHILD2"/>
      <xs:choice>
        <xs:element ref="CHILD3" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="CHILD4" minOccurs="1" maxOccurs="unbounded"/>
      </xs:choice>
      <xs:element ref="CHILD5"/>
    </xs:sequence>
    <xs:attribute name="ID" type="xsd:id"/>
  </xs:complexType>
</xs:element>
```

The Language Definition clause contains syntax diagrams for each element. Here is an example syntax diagram with annotations:
Figure 1 — Example element syntax diagram

Non-normative examples are included in separate clauses, and are shown in this document using a separate font and background:

```xml
<Example attribute1="example attribute value">
  <Element1>example element content</Element1>
</Example>
```

C.2.2 Namespace prefix conventions

This document makes use of vocabularies from several XML namespaces (where the definition of an XML namespace is as specified in W3C XMLNAMES). Qualified Names are written with a namespace prefix followed by a colon followed by the local part of the Qualified Name as shown in the following example.

EXAMPLE  

```
  bbl:bbl
```

For the purposes of this document the Table below gives the namespace prefixes associated with the identified namespaces.

<table>
<thead>
<tr>
<th>Namespace prefix</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbl</td>
<td>urn:mpeg:mpeg21:2006:01-BBL-NS</td>
</tr>
<tr>
<td>did</td>
<td>urn:mpeg:mpeg21:2002:02-DIDL-NS</td>
</tr>
</tbody>
</table>
C.3 Bitstream Binding Language

C.3.1 Introduction

The purpose of this clause is to describe the syntax and semantics of the Bitstream Binding Language (BBL). The syntax is defined using XML schema (as specified in W3C XMLSCHEMA). The goal is to be as flexible and general as possible, so that it may be used for the description of streaming instructions for a Digital Item no matter what its content. For the purposes of this document, the XML schema syntax descriptions are collectively referred to as BBL schema.

C.3.2 Overview

C.3.3 BBL

The Bitstream Binding Language (BBL) enables the description of how a Digital Item – comprising XML and binary content – can be fragmented and inserted (bound) into one of several transport streams (Figure 2). Just as a Digital Item is generic insofar as the XML that is stored in or referenced from the DID and the types of resources that may be attached, BBL is agnostic to the format of the data it is able to process, as well as the type of transport streams to be output.
In this way, BBL provides an abstraction layer between a stored Digital Item and its representation in a specific channel. This enables different bitstreams to be created from a single DI to provide

- different views/subsets of the DI,
- different renderings of the content, and
- different bitstream formats,

facilitating a Universal Multimedia Access (UMA) solution to the serialization of Digital Items.

Different parts of a DI can be sent over separate channels.

**EXAMPLE** In the internet domain, portions of the DI requiring reliable delivery can be sent using, for example, TCP, while others which rely on timely delivery can be sent via RTP.

In BBL, each channel to be used is associated with a Handler (see section C.3.7.16). A BBL processor will provide in a handler the functionality required for operating the transport channel with which it is associated. Additionally, using a BS Schema, declarative behaviour for any particular channel (such as the contents of header fields) can be specified as part of the BBL.

BBL is designed for maximum reusability, so that (in the example, Figure 2) the binding of resources of type Y to output streams of type O can be specified once, and referenced by all DIs using Y and O. In the same way, when metadata is presented in a standard format (Scheme X) for a group of Digital Items, a single BBL binding can be written for that metadata format and applied to all of the DIs.
C.3.4 BBL Documents

There are two basic types of BBL document. The first is an Instance (see C.3.7.9), which describes the streaming instructions for a specific Digital Item, and contains references to the DID and any other resources of the DI (these references are either URIs, or pointers to the location of the URI within the DID or other document).

The second type of BBL document is a Binding (see C.3.7.10), which is an abstract mapping from some Digital Item or part thereof to a particular set of output streams. Bindings are instantiated as part of an Instance document, which provides the URI references for the concrete content to which the Binding(s) are to be applied.

As shown above (Figure 2), a Binding could be written which maps all resources of a particular format (eg. AAC) to a particular output stream (such as RTP). While in this case such mappings might already exist (ie RFC3640 [1]), a BBL Binding provides a programmatic method for specifying a mapping such that a BBL processor is able to directly transmit the desired content. In this way, as new mappings are developed, they can be created once and used by any BBL processor.

C.3.5 Processing of Binary Resources

The Bitstream Syntax Description Language (BSDL, see part 7 of ISO/IEC 21000) is used by BBL to provide an abstraction layer that allows binary resources to be handled in BBL in exactly the same way as XML.

EXAMPLE If the author of a BBL document wishes to include an MPEG-4 Video Elementary Stream VOL object in a particular fragment, they do so using an W3C XPATH expression (as shown, Figure 3). The BBL engine has been previously informed that the mp4 prefix is bound to a binary namespace (and associated with a referenced BS Schema), and so invokes the BSDL parser on the currently active binary object to output the appropriate binary data.

Figure 3 – XML-based selection of binary content
If required, pre-existing BS Descriptions can be treated in a similar manner, by declaring the prefix(es) used in the BSD as binary namespaces. This process is discussed in detail in section C.3.7.23.

### C.3.6 BBL Processing Order

Figure 4 shows the order in which `packet` or `packetStream` elements shall be processed. For each `packet` or `packetStream`:

- a) include elements are resolved (and possibly fragmented, see C.3.7.23),
- b) variable definitions and assigns are processed (see C.3.7.12 and C.3.7.13),
- c) value-of and timing elements are resolved (see C.3.7.33 and C.3.7.24 to C.3.7.26),
- d) BSD content is parsed,
- e) content may optionally be encoded.

**NOTE** MPEG does not currently mandate any normative encoders for use at this step.

- f) the resulting packet content is passed to the handler which outputs the transport stream (see C.3.7.16).

![Figure 4 – BBL Packet Processing Order](image)

It is important to note that value-of and timing are resolved after the variables have been processed, so that these elements can utilise variable values from the current packet.

### C.3.7 Bitstream Binding Language Definition

#### C.3.7.1 Introduction

The following subclauses provide a formal definition of the syntax and semantics for the BBL Elements, Attributes, and provides requirements relating to the use of W3C XPATH expressions within BBL attributes.
C.3.7.2 Validation

Validating a document against the BBL schema (as specified in W3C XMLSCHEMA) is necessary, but not sufficient, to determine its validity with respect to BBL. After a document is validated against the BBL schema, it shall also be subjected to additional validation rules. These additional rules are given below in the descriptions of the elements to which they pertain.

NOTE While validation is required to ensure a valid BBL document is being used, an implementation could validate the document once and it can then use that document without needing to validate it. Re-validation would only be required if the document gets modified.

C.3.7.3 Document Modularity

Modularity of BBL documents can be achieved by utilising XML Inclusions (XInclude) as specified in W3C XINCLUDE. XInclude can be used to reference elements within a document, or to elements in a different document. The former type of reference is known as an internal reference; the latter is known as an external reference. An internal reference allows a single source to be maintained for an element that occurs in more than one place in a BBL document. An external reference allows a BBL document to be split up into multiple linked discrete documents.

C.3.7.4 Use of “?” to delimit XPath expressions within attribute values

Where data is provided to a BBL description that already exists within another document – for example a URI reference to a binary or xml resource – a mechanism is required to allow the field within the BBL to contain a pointer to the actual value. For this purpose, W3C XPATH expressions delimited by “?” symbols may be used in certain attribute or element values which otherwise expect a literal value. These attributes are:

- xmlSource (see C.3.7.6)
- binarySource (see C.3.7.6)
- binding (see C.3.7.10)
- bsSchemaLocation (see C.3.7.17)
- The children of a handler or handlerParams element (see C.3.7.16 and C.3.7.20)

See Figure 5 for an example of the use of delimited W3C XPATH expressions. In the example, the value of the timeBase parameter is found by evaluating the variable $rtp:timeBase.

C.3.7.5 Use of variables and functions within XPath expressions

W3C XPATH supports the use of variables and custom-defined functions. These features can dramatically improve the efficiency of BBL processing, as commonly used values (for example a timebase) may be stored
in a variable once, then subsequent accesses to the variable value are several orders of magnitude faster than repeated resolution of node-test expressions.

BBL provides a normative variable which is used to access data from the underlying processing model so that it may be included in output packets. This variable is specified below.

**Table 3 – BBL XPath Variables**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$bbl:addr</td>
<td>When used within an attribute, this variable will evaluate to an absolute XPath address of the element to which the attribute is attached in the XML source Document from which the element was extracted. This variable is designed to be used with the <code>bbl:context</code> attribute to provide an MPEG-21 receiving peer with the ability to reconstruct a DI from its fragments.</td>
</tr>
</tbody>
</table>

The scope of any variable is always from the point at which it is defined to the end of the containing instance/binding element.

BBL also provides several normative functions for accessing data from the processing model. These are:

**Table 4 – BBL XPath Functions**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bbl:delTime(id)</code></td>
<td>Returns a float representing the Delivery Time of the packet or packetStream referenced by the id corresponding to the value of the id parameter. In the case of a packetStream, the Delivery Time of the first packet in the stream is returned.</td>
</tr>
<tr>
<td><code>bbl:duration(id)</code></td>
<td>Returns a float representing the Duration of the packet or packetStream referenced by the id corresponding to the value of the id parameter. In the case of a packetStream, the Duration of the most recently transmitted packet is returned.</td>
</tr>
</tbody>
</table>
| `bbl:finalDelTime(id)` | Returns a float representing the Delivery Time of the packet or packetStream referenced by the id corresponding to the value of the id parameter. In the case of a packet, the this function returns a value identical to that returned by `bbl:delTime()`.
These functions are particularly useful to specify the Delivery Time of a Packet or PacketStream relative to the Delivery Time of another Packet/PacketStream.

C.3.7.6 xmlSource and binarySource Attributes

BBL provides a hierarchical mechanism for declaring Source Documents identical to that used by W3C XMLNAMES. At any element, the in-scope XML or Binary Source Document is that referenced by the first xmlSource or binarySource attribute (respectively) encountered by searching the ancestor-or-self axis of the element (see W3C XPATH), in order of decreasing proximity to the element. (ie. first the element itself, then its parent, then its grandparent, and so on to the root node).

When W3C XPATH expressions are encountered within an xmlSource attribute, they are resolved as above, with the exception that the search proceeds along the ancestor axis of the element, in order to prevent a circular reference.

Because bindings (see C.3.7.10) are abstract documents, they are not required to declare Source Documents (although they may). When a binding is instantiated by a bind element, the in-scope Source Documents are resolved as if the binding and its subtree were attached to the bind as a child.

Validation Rule:

— Any attribute which takes as its value an W3C XPATH expression must have an in-scope XML Source Document, unless
— the W3C XPATH expression does not contain any node-test elements, or
— the attribute is part of an element which has a binding ancestor.

EXAMPLE This example (see Figure 5) illustrates the behavior of xmlSource and binarySource. Each W3C XPATH expression is indicated with a numerical value. The contexts of these expressions are listed below.

a) Because this W3C XPATH expression is contained within an xmlSource attribute, it is resolved against the XML document referenced by the first xmlSource attribute encountered when searching the ancestors of this node. In this case this is did_foreman.xml which is declared by the <instance> element.

b) This expression is resolved against the XML document referenced by the first xmlSource attribute encountered when searching first the node itself, and then the ancestors of the node. Here, this is the document whose URI is the value returned by a).

c) The XML document against which this expression is resolved is that declared on the element (otherDoc.xml). However, the expression in fact makes no reference to any node-tests, and so the context has no effect.

d) The XML document against which this expression is resolved is did_foreman.xml.
e) Because avi has been declared as a bsd namespace, this W3C XPATH expression is resolved against the binary
document content/foreman.cmp. This is done by inferring the necessary BSD for the binary content using the
declared schema (whose location was resolved by the W3C XPATH expression in b)).

f) The XML document against which this expression is resolved is found by attaching the parent binding element to
the bind from which it is referenced. It is thus did_foreman.xml.

```xml
<instance xmlSource="did_foreman.xml">
  <!--...-->
  <register xmlSource="?//did:Descriptor[@id='constants']/did:Statement/text()?">
    <!--...-->
    <bsd prefix="avi" bsSchemaLocation="?fn:substring-after(/@xsi:SchemaLocation,' ')?"/>
    <handler id="RTP"/>
    <Hrtp:timeBase id="RTP" xmlSource="otherDoc.xml">?$rtp:timeBase?<Hrtp:timeBase/>
    </handler>
  </register>
  <!--...-->
  <packet binarySource="content/foreman.cmp">
    <content>
      <include ref="/" depth="3">
        <include ref="avi:File//avi:Chunk[@id='0f13c']"/>
      </include>
    </content>
  </packet>
  <packetStream delTime="5.0">
    <bind binding="#BINDING_A"/>
  </packetStream>
</instance>

<binding id="BINDING_A">
  <packet>
    <content>
      <include ref="/did:DIDL/did:Item" depth="4"/>
    </content>
  </packet>
</binding>
```

Figure 5 – BBL fragment – xmlSource, binarySource example (informative)
C.3.7.7 sourceElt Type

xmlSource and binarySource attributes may be declared on most BBL elements (see clauses C.3.7.9 to C.3.7.34). They are defined by the abstract complexType sourceElt, from which the relevant BBL elements are extended.

Validation Rule:

— If the binarySource attribute is present, the bSrcPrefix attribute must also be present, and visa-versa.

<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td>Specifies the XML Source Document to be used by W3C XPATH expressions on this element and its children. The value of this attribute may be either a relative or absolute URL, or a “?” delimited W3C XPATH expression which resolves to a URL.</td>
</tr>
<tr>
<td></td>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td>Specifies the Binary Source Document to be used by elements of bs1:byteRange or bs1:bitstreamSegment type that are descendants of this element (whether declared within the BBL document or included). The value of this attribute may be either a relative or absolute URL, or a “?” delimited W3C XPATH expression which resolves to a URL.</td>
</tr>
<tr>
<td></td>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td>Specifies the Namespace Prefix with which the root element of the BS Schema describing the binarySource is associated. This prefix must be registered as being associated with a BSD Namespace (see C.3.7.17). If the binarySource attribute is present, this attribute must also be present.</td>
</tr>
</tbody>
</table>

[source] <xs:complexType name="sourceElt" abstract="true">
  <xs:attribute name="xmlSource" type="xs:string" use="optional"/>
  <xs:attribute name="binarySource" type="xs:string" use="optional"/>
  <xs:attribute name="bSrcPrefix" type="xs:NCName" use="optional"/>
</xs:complexType>
C.3.7.8 bbl Element

The bbl element is the root element of any BBL description. It contains one or more instance and/or binding elements.

The bbl element does not define any processing. All processing is defined by the children of the bbl element.

```
<xs:complexType>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="instance" type="bbl:instance"/>
    <xs:element name="binding" type="bbl:binding"/>
  </xs:choice>
</xs:complexType>
```

C.3.7.9 instance Element

The instance element declares a BBL Instance Document. Instance Documents describe the Fragmentation and binding of concrete Source Document(s), and may instantiate BBL bindings to do so.

Instance documents provide a scope for declared Variables and Handlers.

Validation rule:

- declarations and variables elements may appear zero or one times. register must appear once.
  The three elements may appear in any order, but should be declared so that variables are declared before they are referenced in the document.
```xml
<xs:complexType name="instance">
  <xs:complexContent>
    <xs:extension base="bbl:bblDoc"/>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="bblDoc">
  <xs:complexContent>
    <xs:extension base="bbl:idElt"/>
  </xs:complexContent>
</xs:complexType>
```

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>xs:ID</td>
<td>optional</td>
<td>Optional attribute to identify this element.</td>
</tr>
</tbody>
</table>

**Diagram**

- `bbl:declarations`
- `bbl:variables`
- `bbl:register`
- `bbl:packet`
- `bbl:packetStream`

**Children**

- `bbl:declarations`
- `bbl:variables`
- `bbl:register`
- `bbl:packet`
- `bbl:packetStream`
C.3.7.10 binding Element

The binding element declares a BBL binding document. Binding documents describe an abstract Fragmentation and binding process which may be applied to multiple source documents.

A binding is instantiated using a bind element.

Binding Documents provide a scope for declared variables and handlers.

For an example of the use of this element, see C.3.7.6.

Validation rule:

— declarations and variables elements may appear zero or one times. register must appear once.
C.3.7.11 variables Element

The variables element contains variable definition and assignment instructions.

A variables element may appear as a child of instance, binding, packet or packetStream.

If the variables element is the child of an instance or binding element, then it is processed at time 0. If, however, if it is the child of a packet or packetStream, the variable definitions and assignments are processed at the same time as the packet is processed.
C.3.7.12 define Element

The `define` element allows a variable to be declared so that it may be used in subsequent W3C XPath expressions. Variables are declared by providing

- a variable name as a QName,
- an initial value, which will be equal to the result of the W3C XPath expression contained by the `value` attribute, and
- optionally a type, which shall be one of the following:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:integer</td>
<td>xs:int</td>
</tr>
<tr>
<td>xs:decimal</td>
<td>xs:float</td>
</tr>
<tr>
<td>xs:byte</td>
<td>xs:QName</td>
</tr>
<tr>
<td>xs:hexBinary</td>
<td>xs:unsignedInt</td>
</tr>
<tr>
<td>xs:time</td>
<td>xs:dateTime</td>
</tr>
</tbody>
</table>

The location in time of a `define` element is specified by its parent `variables` element. See C.3.7.11.

Validation Rule:

- For all variables other than those declared by clause C.3.7.5, a `define` element must precede in time any reference to the variable by an `assign` element or W3C XPath expression.

- The value of the name attribute shall not match any of those defined in clause C.3.7.5.

<table>
<thead>
<tr>
<th>diagram</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmlSource</code></td>
<td>xs:string</td>
<td>optional</td>
<td>See sourceElt</td>
<td></td>
</tr>
<tr>
<td><code>binarySource</code></td>
<td>xs:string</td>
<td>optional</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bSrcPrefix</code></td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bbl:name</code></td>
<td>xs:QName</td>
<td>required</td>
<td>The name of the variable to be defined.</td>
<td></td>
</tr>
<tr>
<td><code>bbl:value</code></td>
<td>xs:string</td>
<td>required</td>
<td>An W3C XPath expression which resolves to the value of the variable. The returned value must match the variable type. The context of the W3C XPath expression varies according to the location of the containing variables element. If the variables element is declared by an</td>
<td></td>
</tr>
</tbody>
</table>
instance or binding element, then the context of the W3C XPATH expression is the root node of the in-scope XML Source Document.

If the containing variables element is declared by a packet or packetStream element, then the context is the root node of the content of the current packet.

This is particularly important in the case of a packetStream, where the content of the packet will be a subset of that returned by the include statement according to the declared Fragmentation rules. See C.3.7.23 for further information.

<table>
<thead>
<tr>
<th>type</th>
<th>xs:QName</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>The data type of the variable to be defined. The value of the type attribute must match one of the above list of types.</td>
</tr>
</tbody>
</table>

```xml
<source>
<xs:complexType name="define">
  <xs:complexContent>
    <xs:extension base="bbl:abstractVar">
      <xs:attribute name="type" type="xs:QName" use="optional" default="xs:int"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="abstractVar">
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute ref="bbl:name" use="required"/>
      <xs:attribute ref="bbl:value" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:attribute name="name" type="xs:QName"/>
<xs:attribute name="value" type="xs:string"/>
</source>
```

### C.3.7.13 assign Element

The assign element assigns a new value to any previously declared variable. This is done by providing the name of the variable, a W3C XPATH expression which resolves to the new value.
The location in time of an assign element is specified by its parent variables element. See C.3.7.11.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td>See sourceElt</td>
</tr>
<tr>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>bbl:name</td>
<td>xs:QName</td>
<td>required</td>
<td>See define</td>
</tr>
<tr>
<td>bbl:value</td>
<td>xs:string</td>
<td>required</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<xs:complexType name="abstractVar">
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute ref="bbl:name" use="required"/>
      <xs:attribute ref="bbl:value" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

C.3.7.14 declarations Element

The Declarations element is used to define a set of BBL or other elements - without instantiating them - for later use in a document via an internal reference (see C.3.7.3).

```
<xs:complexType name="declarations">
  <xs:sequence maxOccurs="unbounded">
    <xs:any namespace="##any" processContents="skip"/>
  </xs:sequence>
</xs:complexType>
```

C.3.7.15 register Element

The register element contains all of the Handler and BSD namespace declarations for an instance or binding.
C.3.7.16 handler Element

A `handler` element registers a BBL Handler which represents a particular transport mechanism in BBL.

A BBL Handler receives the byte content of each packet output from the BBL process along with the delivery time and repeat parameters specified for the packet. The Handler is then responsible for transmitting the each packet at the correct time(s) according to the rules of the associated transport mechanism.

Valid Handler types are specified in clause C.4.
attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>xs:ID</td>
<td>optional</td>
<td>The Handler ID. BBL elements make reference to this id to indicate that it is to be used to handle that element.</td>
</tr>
<tr>
<td>type</td>
<td>xs:anyURI</td>
<td>required</td>
<td>The URI identifying the handler to be used. This must correspond to one of the handlers specified in clause C.4.</td>
</tr>
</tbody>
</table>

source

```xml
<xs:complexType name="handlerType">
  <xs:complexContent>
    <xs:extension base="bbl:paramsElt">
      <xs:attribute name="type" type="xs:anyURI" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```xml
<xs:complexType name="paramsElt">
  <xs:complexContent>
    <xs:extension base="bbl:idElt">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:any namespace="##other"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

C.3.7.17  bsd Element

The `bsd` element is used to declare that a prefix used in the BBL Document belongs to a BSD Namespace. Elements with this namespace, whether part of the BBL Document or included by include elements, will be converted to their binary representation by an ISO/IEC 21000-7 BSDToBin process, according to the BS Schema identified by `bsSchemaLocation`.

Validation Rule:

- Prefixes declared by `bsd` elements must be associated with a namespace in the manner prescribed by W3C XMLNAMES (ie an `xmlns` attribute) and must be in scope at the `bsd` element.
### attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>xs:NCName</td>
<td>required</td>
<td>The prefix which is to be declared as belonging to a BSD Namespace.</td>
</tr>
<tr>
<td>bsSchemaLocation</td>
<td>xs:anyURI</td>
<td>required</td>
<td>A URI which corresponds to the location of the BS Schema associated with this BSD Namespace.</td>
</tr>
<tr>
<td>fromBinarySource</td>
<td>xs:boolean</td>
<td>optional</td>
<td>If true or omitted, W3C XPATH expressions containing this namespace are to be resolved against the in-scope Binary Source document. If false, W3C XPATH expressions are resolved against the in scope XML Source and subsequently converted to binary.</td>
</tr>
</tbody>
</table>

```xml
<source>
  <xs:complexType name="bsd">
    <xs:attribute name="prefix" type="xs:NCName" use="required"/>
    <xs:attribute name="bsSchemaLocation" type="xs:anyURI" use="required"/>
    <xs:attribute name="fromBinarySource" type="xs:boolean" use="optional" default="true"/>
  </xs:complexType>
</source>
```

### C.3.7.18 packet Element

A `packet` element contains instructions for assembling a single Packet. It contains a `content` element, which describes how the packet content is assembled, an optional `variables` element, which allows variable values to be updated, and attributes to declare the Delivery Time, Delivery Condition, Repeat Period, and Handler associated with the Packet.

Along with the `packetStream` element, `packet` is the central structure used by BBL to declare streaming instructions.

**Validation Rule:**

- The ID referenced by the `handler` attribute must be attached to a `handler` element within the same instance or binding as this element.
attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>xs:ID</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>params</td>
<td>xs:normalizedString</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td>bbl:handler</td>
<td>xs:IDREF</td>
<td>optional</td>
<td>A reference to the Handler to be used to transmit the packet. If this element is missing, the first handler declared within register will be used.</td>
</tr>
<tr>
<td>delCondition</td>
<td>xs:string</td>
<td>optional</td>
<td>An W3C XPATH expression which evaluates to a boolean. The associated packet will not be delivered until/unless this W3C XPATH expression evaluates to true.</td>
</tr>
<tr>
<td>delTime</td>
<td>xs:string</td>
<td>optional</td>
<td>An W3C XPATH expression which is resolved against the in-scope source document, and evaluates to a number indicating the first time for the packet to be delivered (in seconds) relative to the beginning of the BBL session. If the attribute is not provided, then the delivery time is specified by non-normative means (for example, in a live streaming scenario, packets may be output as soon as input data for them is available).</td>
</tr>
<tr>
<td>rpt</td>
<td>xs:string</td>
<td>optional</td>
<td>An W3C XPATH expression which is resolved against the in-scope source document, and evaluates to the period of repetition (in seconds) for transmission of the packet(Stream). A value of 0 indicates no repetition and is the default.</td>
</tr>
</tbody>
</table>

source

```xml
<xs:complexType name="packet">
  <xs:complexContent>
    <xs:extension base="bbl:abstractPkt">
      <xs:sequence>
        <xs:element name="content" type="bbl:content"/>
        <xs:element name="variables" type="bbl:variables" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>`
C.3.7.19 packetStream Element

A packetStream element is used to declare a Stream of Packets. A packetStream has the same attribute parameters and variables element as packet. However, the content of a packetStream is described using either a contentTemplate, which provides instructions for assembling the content using static and dynamic elements, or a bind element, which instantiates a binding declared elsewhere using the in-scope Source Documents.

Along with the packet element, packetStream is the central structure used by BBL to declare streaming instructions.

Validation Rule:

— The ID referenced by the handler attribute must be attached to a handler element within the same instance or binding as this element.
<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td>See sourceElt</td>
</tr>
<tr>
<td></td>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>xs:ID</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>params</td>
<td>xs:normalizedString</td>
<td>optional</td>
<td>See handler</td>
</tr>
<tr>
<td></td>
<td>bbl:handler</td>
<td>xs:IDREF</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bbl:encoding</td>
<td>xs:IDREF</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delCondition</td>
<td>xs:string</td>
<td>optional</td>
<td>See packet</td>
</tr>
<tr>
<td></td>
<td>rpt</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delTime</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
</tbody>
</table>

The Delivery Time of subsequent packets in the stream is defined by delTimes and/or durations (refer clauses C.3.7.25 and C.3.7.26).

```xml
<xs:complexType name="packetStream">
  <xs:complexContent>
    <xs:extension base="bbl:abstractPkt">
      <xs:sequence>
        <xs:choice>
          <xs:element name="contentTemplate" type="bbl:abstractContent"/>
          <xs:element name="bind" type="bbl:bind"/>
        </xs:choice>
        <xs:element name="variables" type="bbl:variables" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
C.3.7.20 handlerParams Element

The handlerParams element contains parameters which are sent to the handler associated with the parent packet or packetStream. The descendant content of handlerParams is provided to the handler at the same time as the parent packet or packetStream content.

The syntax and semantics of the descendant content of handlerParams is determined by the handler type, as specified by the handler attribute of the parent packet/packetStream (or the default handler if not present).

```
<xs:complexType name="paramsElt">
  <xs:complexContent>
    <xs:extension base="bbl:idElt">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:any namespace="##other"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

C.3.7.21 content Element

The content element contains instructions for assembling the content of a packet. There are five ways that this may be conducted:

- g) Elements from namespaces other than that of BBL may be directly instantiated as descendants of the content element. This includes elements from namespaces which have been declared as BSD.

- h) Elements which are instantiated in other XML Documents may be retrieved by an include element and included in the content, either as a child of content or as a child of any of its descendants.

- i) Structures from Binary Documents which are identified within the BS Schema identified with a BSD Prefix may be retrieved by an include element and included in the content, either as a child of content or as a child of any of its descendants.
j) Attributes may be attached to any element within the content by the use of an attribute element with a value-of child. Where attributes are to be attached to elements which are to be retrieved by an include element, the match attribute may be used to indicate the elements to which the attribute is to be attached.

k) Element content may be added to any element within the content by the use of a value-of element.

<table>
<thead>
<tr>
<th>diagram</th>
<th>bbl:abstractContent</th>
</tr>
</thead>
<tbody>
<tr>
<td>bb:content</td>
<td>any ##any</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
<td>See sourceElt</td>
</tr>
<tr>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>xs:ID</td>
<td>optional</td>
<td></td>
<td>See instance</td>
</tr>
</tbody>
</table>

```xml
<source>
  <xs:complexType name="abstractContent">
    <xs:complexContent>
      <xs:extension base="bbl:idElt">
        <xs:sequence maxOccurs="unbounded">
          <xs:any namespace="##any" processContents="skip"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</source>
```

**C.3.7.22 contentTemplate Element**

The contentTemplate element contains instructions for assembling the content of a Stream of Packets. This is conducted in the same manner as a content element, with the exception that include descendants may have additional semantics to fragment their returned content.
C.3.7.23 include Element

The include element is used to retrieve fragments of external Documents. Its semantic is similar to the XInclude element, however the BBL include has additional functionality tailored to BBL.

The include element contains a ref attribute, whose value is W3C XPATHan XPath 2.0 [3] expression which resolves to a node-set that represents the root elements of the fragments to be included. The optional depth attribute subsequently selects a number of levels of descendants of the node-set to be included. The nodes within the node-set are guaranteed to be in the original document order.

include elements may be nested – in which case the fragments returned by the inner include are attached to those of the outer include in their original location within the Source Document. include elements may be nested to multiple levels and with multiple include elements at each level.

include elements may be used to retrieve fragments of Binary Documents. This is done by referencing elements with a prefix defined as belonging to a BSD namespace (see C.3.7.17). In such a case the context of the W3C XPATH expression is the BS Description which would be created by applying the referenced BS Schema to the in-scope Binary Source Document. Note that the BS Description need not be actually instantiated, its contents may be inferred from the Binary Source Document directly, using the BS Schema.

As well as child include elements, an include element may have zero or more attribute element children. These contain a match attribute to attach attribute(s) to one or more nodes returned by the include element. If the include element is within a contentTemplate, it may also have timing and fragmentation children.

Fragmentation process

In processing include elements within a contentTemplate, the following algorithm is applied:
For each element returned by the include (taking into account both the W3C XPATH expression and depth attribute):

1. Determine the Duration and/or Delivery Time of the element from timing parameters.

If the element matches a "first" constraint, begin a new Packet before adding the element.

If the element matches a "last" constraint, begin a new Packet after the element and its descendants have been processed.

If the element matches an "unbroken" constraint, apply the Fragmentation rules to the subtree of the element, to the depth indicated. If the rules are met, insert the subtree into the current Packet, else insert it into the following Packet.

2. Else, calculate whether the content from this include element within the current Packet will meet the fragmentation rules if the current element is added. If so, add the element, otherwise begin a new Packet and add the element to it.

Validation Rules:

— Because include elements are always descendants of a content or contentTemplate element (which have processContents set to "skip"), an include element will never be validated directly by an XML Schema Validator. However, include elements must still be schema-valid.

— An include shall declare a match attribute only if it is the child of another include element.

— An include element that is the child of another include element shall not declare an xmlSource attribute unless it also declares a match attribute. (The insertion point for the nested include element would become undefined).

— An include element shall declare timing and fragmentation children only if it is the descendant of a contentTemplate element.

— If an include element is the descendant of a contentTemplate element, it shall not contain child include elements.
children  bbl:timing  bbl:fragmentation  bbl:attribute  bbl:include

<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ref</td>
<td>xs:string</td>
<td>required</td>
<td></td>
</tr>
<tr>
<td></td>
<td>depth</td>
<td>xs:integer</td>
<td>optional</td>
<td>The number of levels of descendants of the nodes returned by ref which are to be included. The default value is 0.</td>
</tr>
<tr>
<td></td>
<td>match</td>
<td>xs:string</td>
<td>optional</td>
<td>An W3C XPATH expression which is evaluated against the nodes returned by the parent include element and resolves to the node(s) to which the included content is to be attached. If this attribute is not provided, the included content is attached to the parent content as it was in the source document.</td>
</tr>
</tbody>
</table>

source

```xml
<xs:complexType name="include">
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:sequence>
        <xs:element name="timing" type="bbl:timing"
                     minOccurs="0"/>
        <xs:element name="fragmentation"
                     type="bbl:fragmentation"
                     minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="bbl:attribute"/>
          <xs:element name="include" type="bbl:include"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="ref" type="xs:string"
                     use="required"/>
      <xs:attribute name="depth" use="optional"
                     default="0"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
C.3.7.24 timing Element

A timing element contains delTimes and durations declarations which are used to attach delivery times and durations (respectively) to fragments returned by the parent include element as part of a contentTemplate.

```
<xs:complexType name="timing">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="delTimes" type="bbl:timingElt"/>
    <xs:element name="durations" type="bbl:timingElt"/>
  </xs:choice>
</xs:complexType>
```

C.3.7.25 delTimes Element

A delTimes element is used to attach a Delivery Time to packets containing elements which match a certain W3C XPATH expression.

```
attributes
Name    Type      Use        Semantic
xmlSource xs:string optional    See sourceElt
binarySource xs:string optional
bSrcPrefix xs:NCName optional
match    xs:string optional    An W3C XPATH expression which returns the node-set of elements which are to be operated on by this element. The context of the expression is a Document containing the node-set returned by the parent
```
include element. If the attribute is not present, all elements returned by the parent include element are matched.

<table>
<thead>
<tr>
<th>value</th>
<th>xs:string</th>
<th>required</th>
</tr>
</thead>
</table>

An W3C XPATH expression to be evaluated against each node returned by the match expression, which resolves to the value (in seconds since the beginning of the packetStream) of the delivery time of that node. The context of the expression is each node that it is evaluated against.

```xml
<source>
  <xs:complexType name="timingElt">  
    <xs:complexContent>  
      <xs:extension base = "bbl:sourceElt">
        <xs:attribute ref = "bbl:match " use = "optional "/>
        <xs:attribute ref = "bbl:value " use = "required "/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:attribute name="match" type="xs:string" default="."/>
</source>
```

### C.3.7.26 durations Element

A durations element is used to attach Durations to elements returned by an include element which match a certain W3C XPATH expression.

<table>
<thead>
<tr>
<th>attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xmlSource</td>
<td>xs:string</td>
<td>optional</td>
<td>See sourceElt</td>
</tr>
<tr>
<td></td>
<td>binarySource</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bSrcPrefix</td>
<td>xs:NCName</td>
<td>optional</td>
<td>See delTimes</td>
</tr>
<tr>
<td></td>
<td>bbl:match</td>
<td>xs:string</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bbl:value</td>
<td>xs:string</td>
<td>required</td>
<td></td>
</tr>
</tbody>
</table>

An W3C XPATH expression to be evaluated against each node returned by the match expression, which resolves to the value (in seconds) of the duration of that node. The context of the expression is each node that it is evaluated against.

```xml
<xs:complexType name="timingElt">  
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute ref="bbl:match" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
C.3.7.27 fragmentation Element

The fragmentation element contains children which specify the rules for fragmenting the content returned by the parent include element.

If the declared Fragmentation rules are such that any particular element may never be included in a packet (for example an element is constrained to be unbroken but the maximum size is smaller than that of the element), then it shall be included in such a way that the minimum number of rules are violated by the smallest amount (in the example it would be included by itself within a separate packet).

A fragmentation element may contain zero children however such a case has no behaviour.
C.3.7.28 size Element

A size element declares that content should be fragmented so that its size (when serialized and/or binarised) is no greater than the given value.

```xml
<xs:complexType name="size">
  <xs:attribute name="value" type="xs:positiveInteger" use="required"/>
</xs:complexType>
```

C.3.7.29 duration Element

A duration element declares that content should be fragmented so that the total Duration of elements in any one packet is no greater than the given value.

Any element with no Duration attached to it (via the durations element) has a Duration of zero.

```xml
<xs:complexType name="duration">
  <xs:attribute name="value" type="xs:float" use="required"/>
</xs:complexType>
```

C.3.7.30 count Element

A count element declares that content should be fragmented so that the number of nodes matched by the match expression is no greater than the given value.
C.3.7.31 constraint Element

A `Constraint` element applies a constraint to the Fragmentation of nodes which are matched by the `match` W3C XPATH expression. The possible constraints are given in the Table below.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>All matched nodes must be the first content included in any packet. The packet is consequently completed immediately before an element matched with this constraint.</td>
</tr>
<tr>
<td>last</td>
<td>All matched nodes must be the last content included in any packet. The packet is consequently completed immediately after an element matched with this constraint.</td>
</tr>
<tr>
<td>unbroken</td>
<td>All matched nodes must have their entire included subtree included within a single packet.</td>
</tr>
</tbody>
</table>
C.3.7.32 bind Element

A `bind` element is used to instantiate a binding.

The binding to be used may either be specified by the `binding` attribute, attached directly as a child element, or attached via an XInclude statement.

The context of all timing and W3C XPATH expressions within the associated `binding` is relative to those in-scope at the `bind` element. For example, if the `packetStream` in which the `bind` element is contained has `delTime`="10.0", and the referenced Binding contains a `packet` with `delTime`="5.0", then that `packet` will be delivered at time = 15.0 seconds. See C.3.7.6 for an example regarding the context of W3C XPATH statements within bindings.

Validation Rule:

— If the `binding` attribute is present, the `binding` child element shall be absent, and visa versa.

### Table: bbl:binding

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>xmlSource</code></td>
<td><code>xs:string</code></td>
<td>optional</td>
<td>See <code>sourceElt</code></td>
</tr>
<tr>
<td></td>
<td><code>binarySource</code></td>
<td><code>xs:string</code></td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bSrcPrefix</code></td>
<td><code>xs:NCName</code></td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>binding</code></td>
<td><code>xs:string</code></td>
<td>optional</td>
<td>EITHER:</td>
</tr>
</tbody>
</table>

A URL which locates the binding to be attached at this point,
An W3C XPATH expression, delimited by "?", which resolves to a URL locating the binding to be attached at this point.

C.3.7.33 value-of Element

The value-of element has the same semantic as the XSLT [1] element of the same name. It is used to populate the content of elements and attributes declared as part of a content or contentTemplate element, with values retrieved from an external source. While include provides a powerful facility for assembling XML subtrees, value-of is intended to return values of simple type.

The text content of the containing element, or value of the containing attribute is set to the resolved value of the select W3C XPATH expression.
C.3.7.34 attribute Element

An attribute element is used to attach an attribute to an element which is either declared as a descendant of a content or contentTemplate element, or inferred by an include element.

In the former case, the attribute element is declared as a child of the element to which the attribute is to be attached.

EXAMPLE

```xml
<content>
  <did:DIDL>
    <attribute bbl:name="xmlns:bbl">
      <value-of select="bbl:bbl/namespace-uri()" xmlSource="#"/>
    </attribute>
    ...
  </did:DIDL>
</content>
```

In the latter case (where the attribute is to be attached to an element inferred by an include), the attribute element is declared as a child of the include element, and provided with a match attribute which returns the node(s) to which the attribute is to be attached.

EXAMPLE

```xml
<content>
  <include ref="/did:DIDL" depth="2">
    <attribute bbl:name="xmlns:bbl" match=".">
      <value-of select="bbl:bbl/namespace-uri()" xmlSource="#"/>
    </attribute>
    ...
  </include>
</content>
```
Validation Rule:

— An attribute element shall have a match attribute only if the attribute element is the child of an include element.

```
<xs:element name="attribute">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="bbl:idElt">
        <xs:sequence>
          <xs:element ref="bbl:value-of" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="bbl:name" use="required"/>
        <xs:attribute ref="bbl:match" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

C.3.7.35 BBL provided Attributes

The following attributes are normatively specified by BBL, so that an MPEG-21 receiving Peer may reassemble a DID and infer other properties of the packet.

Table 6 – BBL Provided Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbl:context</td>
<td>The value of this attribute is a W3C XPATH expression which resolves to the location at which the parent element (and its subtree) should be</td>
</tr>
</tbody>
</table>
attached to the re-constructed DI.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bbl:rap</code></td>
<td>This attribute has a Boolean value indicating whether the containing packet may be considered to be a Random Access Point. This is to be used when the underlying transport mechanism does not have a facility for signalling such a condition. If a <code>bbl:rap</code> attribute is not attached to a packet, and the underlying stream does not indicate it, the packet is assumed to be not a Random Access Point.</td>
</tr>
<tr>
<td><code>bbl:seqNumber</code></td>
<td>The value of this attribute is an integer indicating the sequence number of the containing packet. It is to be used when the underlying transport mechanism does not provide such a value, and is necessary for computing the Processability of packets in a stream.</td>
</tr>
<tr>
<td><code>bbl:processable</code></td>
<td>This attribute has a Boolean value indicating whether the containing packet is Processable. If the value is false, then the packet shall not be processed until a subsequent packet (as indicated by <code>bbl:seqNumber</code> or the underlying transport mechanism) is received with a <code>bbl:processable</code> attribute set to true. A packet without a <code>bbl:processable</code> attribute is assumed to have such an attribute with a value of true.</td>
</tr>
</tbody>
</table>

### C.4 Definition of handlers for the Bitstream Binding Language

#### C.4.1 Introduction

This clause defines the syntax and semantics of the handlers which are provided for use by the Bitstream Binding Language.

#### C.4.2 MPEG-2 Transport Stream handler

##### C.4.3 Introduction


The output of the handler is Transport Stream packets which can be multiplexed into the output stream by an external system.

##### C.4.4 Handler type Declaration

The MPEG-2 Transport Stream handler is instantiated by registering a handler with a type value of `urn:mpeg:mpeg21:2006:01-BBL-NS:handler:MPEG2TS`
See clause C.3.7.16 for more information.

### C.4.5 Handler Parameter Syntax

The following XML SCHEMA fragment specifies the syntax for parameters which may be instantiated as child elements of a handler with type equal to that specified in C.4.4.

As specified in C.3.7.4, all parameter values may be delimited XPath expressions. Each XPath expression must resolve to a value conforming to the element type.

```xml
<x:simpleType name="hexByte">
    <xs:restriction base="xs:hexBinary">
        <xs:length value="1"/>
    </xs:restriction>
</xs:simpleType>

<x:element name="mode">
    <xs:simpleType>
        <xs:restriction base="Hmpg2:hexByte">
            <xs:enumeration value="15"/>
            <xs:enumeration value="16"/>
            <xs:enumeration value="17"/>
            <xs:enumeration value="18"/>
            <xs:enumeration value="19"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<x:element name="metadataServiceID" type="Hmpg2:hexByte"/>

<x:element name="metadataFormat" type="Hmpg2:hexByte"/>

<x:element name="metadataApplicationFormat">
    <xs:simpleType>
        <xs:restriction base="xs:hexBinary">
            <xs:length value="2"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<x:simpleType name="uint22">
    
</xs:simpleType>
```
C.4.6 Handler Parameter Semantics

Table 7 specifies the semantics for the parameters of the MPEG-2 Transport Stream handler.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hmpg2:mode</td>
<td>required</td>
<td>Specifies the metadata delivery mechanism used according ISO/IEC 13818-1, Table 2.29. Permissible values are 0x15 to 0x19.</td>
</tr>
<tr>
<td>Hmpg2:metadataServiceID</td>
<td>required</td>
<td>Specifies the metadata service id as defined in ISO/IEC 13818-1 clause 2.6.59.</td>
</tr>
<tr>
<td>Hmpg2:metadataFormat</td>
<td>required</td>
<td>Indicates the format and coding of the metadata as specified by ISO/IEC 13818-1</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Hmpg2:metadataApplicationFor</strong></td>
<td>optional</td>
<td>Specifies the application responsible for defining usage, syntax and semantics of the Metadata Pointer and Content Labelling descriptors, as per ISO/IEC 13818-1 clause 2.6.57.</td>
</tr>
<tr>
<td><strong>Hmpg2:metadataInputLeakRate</strong></td>
<td>required</td>
<td>Specifies the input leak rate for the associated metadata as defined in ISO/IEC 13818-1 clause 2.6.63.</td>
</tr>
<tr>
<td><strong>Hmpg2:metadataBufferSize</strong></td>
<td>required</td>
<td>Specifies the size of buffer in the STD model for the associated metadata stream as defined in ISO/IEC 13818-1 clause 2.6.63.</td>
</tr>
<tr>
<td><strong>Hmpg2:metadataOutputLeakRate</strong></td>
<td>optional</td>
<td>Specifies the output leak rate for the associated metadata as defined in ISO/IEC 13818-1 clause 2.6.63. If <code>mode</code> is set to 0x15 or 0x19, this field must not be present; otherwise it is required.</td>
</tr>
<tr>
<td><strong>Hmpg2:contentReferenceID</strong></td>
<td>optional</td>
<td>Specifies the content_reference_id_record as defined in ISO/IEC 13818-1 in clause 2.6.56. If this element is present, <code>content_reference_id_record_flag</code> is set to 1, <code>content_reference_id_record_length</code> is set to a value equal to the number of bytes contained in the element value, and the element value is stored in the required number of <code>content_reference_id_bytes</code>. If this element is absent, <code>content_reference_id_record_flag</code> is set to 0.</td>
</tr>
<tr>
<td><strong>Hmpg2:decoderConfig</strong></td>
<td>optional</td>
<td>Specifies the decoder configuration information as defined in ISO/IEC 13818-1 in clause 2.6.61. If this element is present, then <code>decoder_config_flags</code> shall have a value of 001 and the value of the element stored in the required number of <code>decoder_config_bytes</code>. <code>decoder_config_length</code> shall be set to a value equal to the number of bytes contained in the element.</td>
</tr>
</tbody>
</table>
value.
If this element is absent, decoder_config_flags shall have a value of 000.

Hmpg2:contentTimeBase | required | Specifies the value of content_time_base_indicator as defined in ISO/IEC 13818-1 clause 2.6.57.

Hmpg2:privateData | optional | Specifies private information associated with a metadata service as defined in ISO/IEC 13818-1 clause 2.6.56. If present, the value of this element will be inserted as private_data_bytes.

C.4.7 Real Time Protocol handler

C.4.8 Introduction

The Real Time Protocol (RTP) handler outputs packets as specified by IETF RFC3550.

Packet data passed to the handler is inserted into the RTP payload. RTP header fields are populated generally according to the principals layed down in IETF RFC3550, and specifically as follows:

— Padding, Extension, and CSRC count are set to zero

— Marker and Payload Type are set according to the parameters specified below

— Sequence number is initialised to a random value at the commencement of the session and is incremented by one each time a packet is output

— Timestamp is set to an offset (which is initialised with a random value) plus the delivery time of the packet multiplied by the value of the timebase parameter (below).

— SSRC is assigned to a random value at the commencement of each session

One RTP handler must be instantiated for each individual stream to be transmitted.

C.4.9 Handler type Declaration

The RTP handler is instantiated by registering a handler with a type value of

urn:mpeg:mpeg21:2006:01-BBL-NS:handler:RTP

See clause C.3.7.16 for more information.
C.4.10 Handler Parameter Syntax

The following XML SCHEMA fragment specifies the syntax for parameter elements which may be instantiated as child elements of a handler with type equal to that specified in C.4.9.

```xml
<xs:element name="timeBase" type="xs:unsignedLong"/>
<xs:element name="payloadType" type="xs:unsignedByte"/>
<xs:element name="host" type="xs:normalizedString"/>
<xs:element name="port" type="xs:unsignedShort"/>
<xs:element name="sdp" type="xs:anyURI"/>
<xs:element name="sdpMediaNo" type="xs:unsignedByte"/>
```

The following XML SCHEMA fragment specifies the syntax for parameter elements which may be instantiated as child elements of a handlerParams element for a handler with type equal to that specified in C.4.9.

```xml
<xs:element name="marker">
  <xs:simpleType>
    <xs:restriction base="xs:unsignedByte">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

As specified in C.3.7.4, all parameter values may be delimited XPath expressions. Each XPath expression must resolve to a value conforming to the element type.

C.4.11 Handler Parameter Semantics

Table 7 specifies the elements which may be used to provide parameters to the RTP handler.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Use</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hrtp:timeBase</td>
<td>required</td>
<td>The amount which the timestamp header field is to increment each second.</td>
</tr>
<tr>
<td>Hrtp:payloadType</td>
<td>optional</td>
<td>The value which is to be used for the payload type header field. If the element is absent, the handler will assign a value to the payload type field based on the SDP data.</td>
</tr>
<tr>
<td>Element</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Hrtp:host</td>
<td>optional</td>
<td>The IPv4 address or host string to which the RTP session is to be directed. If this is absent the handler will address packets according to the SDP (if provided), or to the host which has requested the BBL.</td>
</tr>
<tr>
<td>Hrtp:port</td>
<td>optional</td>
<td>The port to which the RTP session is to be directed. If the element is absent, the handler will assign a port based on the SDP data.</td>
</tr>
<tr>
<td>Hrtp:sdp</td>
<td>optional</td>
<td>A URL resolving to an SDP file which is transmitted at the commencement of the session. If <code>payloadType</code> or <code>port</code> are absent, <code>sdp</code> must be present.</td>
</tr>
<tr>
<td>Hrtp:sdpMediaNo</td>
<td>optional</td>
<td>A value between 0 and n-1 (where n is the number of media declarations within the SDP file) indicating the media declaration from which the payload type and/or port are to be drawn. If <code>payloadType</code> or <code>port</code> are absent, <code>sdpMediaNo</code> must be present.</td>
</tr>
<tr>
<td>Hrtp:marker</td>
<td>optional</td>
<td>This element may optionally be provided as the child of a <code>handlerParams</code> element to indicate the value of the marker header field for a Packet or Packet Stream. Its value may 0 or 1. If the element is not present, the marker header field will be zero.</td>
</tr>
</tbody>
</table>
Bibliography


Appendix D

Bitstream Binding Language XML Schema

Listing D.1: XML Schema for the Bitstream Binding Language

```xml
<xs:schema
  xmlns:bbl="urn:mpeg:mpeg21:2006:01–BBL–NS"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:mpeg:mpeg21:2006:01–BBL–NS"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
             schemaLocation="http://www.w3.org/2001/xml.xsd"/>
  <!-- Abstract Type Declarations -->
  <xs:attribute name="name" type="xs:QName">
    <xs:documentation>Specifies the name of a variable, function or attribute.</xs:documentation>
  </xs:attribute>
  <xs:attribute name="value" type="xs:string">
    <xs:documentation>An XPath expression which should resolve to the value to be assigned to a variable or timing element.</xs:documentation>
  </xs:attribute>
  <xs:attribute name="match" type="xs:string" default="."/>
    <xs:documentation>Contains an XPath expression which is resolved against the current xmlSource. Used by timingElt, size, count duration and constraint elements.</xs:documentation>
  </xs:attribute>
</xs:schema>
```
<xs:attribute name="handler" type="xs:IDREF"/>

<!--Declares the handler used to process the output of the associated element. The value of this attribute must match the id of a registered handler.-->

<xs:complexType name="sourceElt" abstract="true"/>

<!--xmlSource attribute declares an XML source document which is used to resolve XPath expressions within the containing element and its descendants. This can be achieved via a URI or via an XPath pointer to a URI (which is enclosed by "?" delimiters). Similarly, binarySource specifies a binary resource which is used in the BSDtoBin process to resolve bitstreamSegments and byteRanges.-->

<xs:complexType name="idElt">
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute name="id" type="xs:ID" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!--Defines the attributes and elements common to all BBL Document types.-->

<xs:complexType name="bblDoc">
  <xs:complexContent>
    <xs:extension base="bbl:idElt">
      <xs:sequence maxOccurs="3">
        <xs:element name="declarations" type="bbl:declarations"/>
        <xs:element name="variables" type="bbl:variables"/>
        <xs:element name="register" type="bbl:register"/>
      </xs:sequence>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="packet" type="bbl:packet"/>
        <xs:element name="packetStream" type="bbl:packetStream"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="abstractVar">
  <xs:annotation>
    <xs:documentation>Defines the attributes used by elements related to variables.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute ref="bbl:name" use="required"/>
      <xs:attribute ref="bbl:value" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="paramsElt">
  <xs:annotation>
    <xs:documentation>The params attribute provides children which are passed to the associated handler. The text content of the children may contain xpath statements, delimited by ? characters, which are evaluated against the current source before being passed to the handler.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:idElt">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:any namespace="##other"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="handlerType">
  <xs:annotation>
    <xs:documentation>Registers a handler.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:paramsElt">
      <xs:attribute name="handlerURI" type="xs:anyURI" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="encoderType">
  <xs:annotation>
    <xs:documentation>Registers an encoder.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:paramsElt">
      <xs:attribute name="encoderURI" type="xs:anyURI" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="functionType">
  <xs:annotation>
    <xs:documentation></xs:documentation>
  </xs:annotation>
</xs:complexType>
<xs:documentation>Defines an ECMAScript based XPath function</xs:documentation>
<xs:complexType>
<!---- >
<xs:complexType name="multiplexerType">
<xs:annotation>
<xs:documentation></xs:documentation>
<xs:documentation></xs:documentation>
<xs:sequence>
<xs:element name="channel" type="bbl:channelType" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="mode" use="required">
<xs:simpleType>
<xs:restriction base="xs:NCName">
<xs:enumeration value="packetCount"/>
<xs:enumeration value="bandwidth"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="weight" type="xs:int" use="optional" default="1"/>
</xs:complexType>
<!---- >
<xs:complexType name="channelType">
<xs:attribute name="id" type="xs:string" use="required"/>
<xs:attribute name="weight" type="xs:int" use="optional" default="1"/>
</xs:complexType>
<!---- >
<xs:complexType name="abstractPkt" abstract="true">
<xs:annotation>
<xs:documentation>Defines the temporal and other parameters common to packet and packetStream elements.</xs:documentation>
</xs:annotation>
<xs:complexContent>
<xs:extension base="bbl:idElt">
<xs:sequence>
<xs:element name="handlerParams" type="bbl:paramsElt" minOccurs="0"/>
</xs:sequence>
<xs:attribute ref="bbl:handler" use="optional"/>
<xs:attribute name="delCondition" type="xs:string" use="optional"/>
<xs:attribute name="delTime" type="xs:string" use="optional"/>
<xs:attribute name="rpt" type="xs:string" use="optional" default="0"/>
<xs:annotation>
<xs:documentation>Defines the period of repetition (in seconds) for transmission of the packetStream. A value of 0 indicates no repetition.</xs:documentation>
</xs:annotation>
</xs:complexContent>
</xs:complexType>
<!---- >
<xs:complexType name="abstractContent"
<xs:annotation>
  <xs:documentation>Defines the parameters common to content and contentTemplate elements.</xs:documentation>
</xs:annotation>

<xs:complexType>
  <xs:complexContent>
    <xs:extension base="bbl:idElt">
      <xs:sequence maxOccurs="unbounded">
        <xs:any namespace="##any" processContents="skip"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="timingElt">
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:attribute ref="bbl:match" use="optional"/>
      <xs:attribute ref="bbl:value" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="instance">
  <xs:complexContent>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="instance" type="bbl:instance"/>
      <xs:element name="binding" type="bbl:binding"/>
    </xs:choice>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="binding">
  <xs:complexContent>
    <xs:extension base="bbl:bblDoc"/>
  </xs:complexContent>
</xs:complexType>

<xs:element name="bbl">
  <xs:annotation>
    <xs:documentation>Document element for Bitstream Binding Language documents.</xs:documentation>
  </xs:annotation>
</xs:element>

<xs:complexType name="instance">
  <xs:annotation>
    <xs:documentation>Declares a BBL instance which operates on concrete documents.</xs:documentation>
  </xs:annotation>
</xs:complexType>

<xs:complexType name="binding">
  <xs:annotation>
    <xs:documentation>Declares an abstract BBL binding which must be instantiated within a BBL instance.</xs:documentation>
  </xs:annotation>
</xs:complexType>
<xs:complexType name="register">
  <xs:annotation>
    <xs:documentation>Registers handlers and other objects for the BBL instance</xs:documentation>
  </xs:annotation>
  <xs:complexType base="bbl:sourceElt">
    <xs:sequence>
      <xs:element name="handler" type="bbl:handlerType" maxOccurs="unbounded"/>
      <xs:element name="encoder" type="bbl:encoderType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="function" type="bbl:functionType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="multiplexer" type="bbl:multiplexerType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="bsd" type="bbl:bsd" minOccurs="0"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:complexType>

<xs:complexType name="bsd">
  <xs:annotation>
    <xs:documentation>All prefixes other than those belonging to the BBL namespace are checked against the set of bsd registrations. If a match is found, elements with the corresponding prefix are parsed using BSDToBin prior to output. schemaLocation attributes are mandatory for any prefixes declared as bsd.</xs:documentation>
  </xs:annotation>
  <xs:sequence maxOccurs="unbounded">
    <xs:any namespace="##any" processContents="skip">
      <xs:annotation>
        <xs:documentation>processing must be skip to allow bbl elements/attributes to be inserted into elements of other namespaces.</xs:documentation>
      </xs:annotation>
    </xs:any>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="declarations">
  <xs:annotation>
    <xs:documentation>Allows the declaration of XML elements without instantiating them, for later use via an xinclude.</xs:documentation>
  </xs:annotation>
  <xs:sequence maxOccurs="unbounded">
    <xs:any namespace="#any" processContents="skip">
      <xs:annotation>
        <xs:documentation>processing must be skip to allow bbl elements/attributes to be inserted into elements of other namespaces.</xs:documentation>
      </xs:annotation>
    </xs:any>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="variables">
  <xs:annotation>
    <xs:documentation>Provides for the declaration and assignment of variables which may be used within xPath expressions.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:choice maxOccurs="unbounded">
        <xs:element name="define" type="bbl:define"/>
        <xs:element name="assign" type="bbl:abstractVar"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="define">
  <xs:annotation>
    <xs:documentation>Defines a variable which may be subsequently used within xPath statements</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:abstractVar">
      <xs:attribute name="type" type="xs:QName" use="optional" default="xs:int"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="packet">
  <xs:annotation>
    <xs:documentation>Declares a single packet</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:abstractPkt">
      <xs:sequence>
        <xs:element name="content" type="bbl:abstractContent"/>
        <xs:element name="variables" type="bbl:variables" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="packetStream">
  <xs:annotation>
    <xs:documentation>Declares a stream of packets</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:abstractPkt">
      <xs:sequence/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:choice>
  <xs:element name="contentTemplate" type="bbl:abstractContent"/>
  <xs:element name="bind" type="bbl:bind"/>
</xs:choice>
<xs:element name="variables" type="bbl:variables" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:complexType>
<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−>
<xs:complexType name="include">
  <xs:annotation>
    <xs:documentation>Includes binary or xml content within a packet or packetStream. Include elements with the same source may be nested, in which case the content of nested includes is inserted into the content returned by the outer include in the same location as it existed in the original document. Note: timing and fragmentation are valid only when the include element is contained within</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:sequence>
        <xs:element name="timing" type="bbl:timing" minOccurs="0"/>
        <xs:element name="fragmentation" type="bbl:fragmentation" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="bbl:attribute"/>
          <xs:element name="include" type="bbl:include"/>
        </xs:choice>
        <xs:element name="events" type="bbl:eventsType" minOccurs="0"/>
        <xs:element name="encode" type="bbl:encode" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" type="xs:string" use="required"/>
      <xs:attribute name="depth" use="optional" default="0">
        <xs:simpleType>
          <xs:restriction base="xs:integer">
            <xs:minInclusive value="−1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute ref="bbl:match" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−>
<xs:complexType name="timing">
  <xs:annotation>
    <xs:documentation>Declares mechanisms for associating timing with content returned by an include element.</xs:documentation>
  </xs:annotation>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="delTimes" type="bbl:timingElt"/>
  </xs:choice>
</xs:complexType>
<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−>
<xs:complexType name="include">
  <xs:annotation>
    <xs:documentation>Includes binary or xml content within a packet or packetStream. Include elements with the same source may be nested, in which case the content of nested includes is inserted into the content returned by the outer include in the same location as it existed in the original document. Note: timing and fragmentation are valid only when the include element is contained within</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="bbl:sourceElt">
      <xs:sequence>
        <xs:element name="timing" type="bbl:timing" minOccurs="0"/>
        <xs:element name="fragmentation" type="bbl:fragmentation" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="bbl:attribute"/>
          <xs:element name="include" type="bbl:include"/>
        </xs:choice>
        <xs:element name="events" type="bbl:eventsType" minOccurs="0"/>
        <xs:element name="encode" type="bbl:encode" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" type="xs:string" use="required"/>
      <xs:attribute name="depth" use="optional" default="0">
        <xs:simpleType>
          <xs:restriction base="xs:integer">
            <xs:minInclusive value="−1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute ref="bbl:match" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−>
delTime and a duration, delTime takes precedence.</xs:documentation>
</xs:element>
</xs:complexType>
<xs:complexType name="fragmentation">

<xs:complexType name="fragmentAt">

<xs:attribute name="id" type="xs:ID" use="optional" default="none"/>
<xs:attribute ref="bbl:match" use="required"/>
<xs:attribute name="delTime" type="xs:string" use="optional"/>
</xs:complexType>
</xs:complexType>
</xs:complexType>
<xs:attribute name="rpt" type="xs:string" use="optional" default="0"/>
</xs:complexType>
<!-- ************************************************************************--->
<xs:complexType name="size">
  <xs:annotation>
    <xs:documentation>Declares the maximum size (in bytes) of packets containing elements matching an XPath pattern</xs:documentation>
  </xs:annotation>
  <xs:attribute name="value" type="xs:positiveInteger" use="required"/>
</xs:complexType>
<!-- ************************************************************************--->
<xs:complexType name="count">
  <xs:annotation>
    <xs:documentation>Declares the maximum count of elements matching an XPath pattern within a single packet</xs:documentation>
  </xs:annotation>
  <xs:attribute name="value" type="xs:positiveInteger" use="required"/>
  <xs:attribute ref="bbl:match" use="optional"/>
</xs:complexType>
<!-- ************************************************************************--->
<xs:complexType name="duration">
  <xs:annotation>
    <xs:documentation>Declares the maximum duration (with value specified according to in context time scheme) of packets containing elements matching an XPath pattern</xs:documentation>
  </xs:annotation>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!-- ************************************************************************--->
<xs:complexType name="constraint">
  <xs:annotation>
    <xs:documentation>Declares constraints on the fragmentation of content. For example, elements may be constrained to appear first or not at all within a packet. Note: it is possible for constraints to be conflicting, for example an element is specified as unbroken whereas one of its descendants is specified as first or last. In this instance, the constraint specified on the node closest to the root will always prevail</xs:documentation>
  </xs:annotation>
  <xs:attribute name="type" use="required"/>
  <xs:simpleType>
    <xs:restriction base="xs:NCName">
      <xs:enumeration value="first"/>
      <xs:enumeration value="last"/>
      <xs:enumeration value="unbroken"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:attribute ref="bbl:match" use="optional"/>
</xs:complexType>
<!-- ************************************************************************--->
<xs:complexType name="encode">
  <xs:complexContent>
    <xs:extension base="bbl:paramsElt">
      <xs:attribute ref="bbl:encoding" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:attribute ref="bbl:match" use="optional"/>
</xs:extension>
</xs:complexType>

<!-- Comment: I think probably we should register event listeners and then reference them by id rather than defining them inline? Or at least allow both options? GD -->

<xs:complexType name="eventsType">
  <xs:sequence>
    <xs:element name="listener" type="bbl:listenerType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<!-- Elts & Attrs used in BBL documents attached to non-bbl elements ** -->

<xs:element name="value-of">
  <xs:annotation>
    <xs:documentation>
      Contains an XPath expression which is resolved against the current xmlSource, the result of which is inserted into the content of the containing element. May only be used on the descendants of declarations, content or contentTemplate.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="bbl:sourceElt">
        <xs:attribute name="select" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="attribute">
  <xs:annotation>
    <xs:documentation>
      Attaches an attribute to all "match"ed elements with a "name" and value equal to the result of the "select" XPath expression in the value-of child. Note: The value expression must evaluate to a simple type or an error will occur.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="bbl:idElt">
        <xs:sequence>
          <xs:element ref="bbl:value-of" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="bbl:name" use="required"/>
        <xs:attribute ref="bbl:match" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:attribute name="encoding" type="xs:IDREF">
  <xs:annotation>
    <xs:documentation>
      Declares the encoder used to process the output of the associated element. The value of this attribute must match the id of a registered encoding.
    </xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute>
  <xs:annotation>
    <xs:documentation>
      May be attached to the root element of an xml packet to denote that the packet represents a random access point — ie that the packet may be processed without knowledge of other packets and that subsequent packets rely only on this and subsequent packets.
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="processable" type="xs:boolean" default="true">
  <xs:documentation>
    May be attached to the root element of an xml packet to denote that the packet is, or is not immediately processable. If the value is false, a BBL receiver will not process the packet until a subsequent packet with processable="true" arrives.
  </xs:documentation>
</xs:attribute>

<xs:attribute name="context" type="xs:string" fixed="$addr">
  <xs:documentation>
    May be attached to any element to indicate the original context of that element. The attribute value is replaced at run time with a xpath address.
  </xs:documentation>
</xs:attribute>
Appendix E

Listings for multi-channel rich media delivery

Listing E.1: BS Schema for MPEG-2 system streams

```xml
<xsd:schema xmlns:bs0="urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS"
xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
xmlns:bs2="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS"
xmlns:mpg2="urn:mpeg2:systems:bitstream"
xmlns:uint="urn:UnsignedIntegers"
xmns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:mpeg2:systems:bitstream"
elementFormDefault="qualified"
bs2:rootElement="mpg2:Bitstream"
<xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
schemaLocation="../../BSDL-1.4.0/schemata/MPEG-B-BSDL-1.xsd"/>
<!-- ************* Root element declaration ************************** -- >
<xsd:element name="Bitstream">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="mpg2:Pack" maxOccurs="unbounded"/>
      <!-- <xsd:element ref="mpg2:TSPacket" maxOccurs="unbounded"-->-->
    </xsd:choice>
    <xsd:attribute ref="bs1:bitstreamURI"/>
  </xsd:complexType>
</xsd:element>
<!-- ************* Pack declaration ************************** -- >
<xsd:element name="Pack" bs2:ifNext="000001BA">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="PackSC" type="xsd:int"/>
      <xsd:element name="null" type="bs1:b2" fixed="1"/>
      <xsd:element name="SCR0" type="bs1:b3"/>
      <xsd:element ref="mpg2:Mkr"/>
      <xsd:element name="SCR1" type="bs1:b15"/>
      <xsd:element ref="mpg2:Mkr"/>
      <xsd:element name="SCR2" type="bs1:b15"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```
<xsd:element ref="mpg2:Mkr"/>
<xsd:element name="SCR3" type="bs1:b9"/>
<xsd:element ref="mpg2:Mkr"/>
<xsd:element name="ProgramMuxRate" type="bs1:b22"/>
<xsd:element ref="mpg2:Mkr"/>
<xsd:element ref="mpg2:Mkr"/>
<xsd:element name="reserved" type="bs1:b5"/>
<xsd:element name="StuffingLength" type="bs1:b3"/>
<xsd:element name="PackStuffing" type="xsd:byte" minOccurs="0"
maxOccurs="unbounded" bs2:nOccurs="mpg2:StuffingLength"/>
<xsd:element ref="mpg2:SystemHeader" minOccurs="0"/>
<xsd:choice maxOccurs="unbounded">
  <xsd:element name="AudioPES" type="mpg2:PESType" bs2:ifNext="000001C0
000001DF"/>
  <xsd:element name="VideoPES" type="mpg2:PESType" bs2:ifNext="000001E0
000001EF"/>
  <xsd:element name="OtherPacket" type="mpg2:MiscPacket"
bs2:ifNext="000001BC 000001FF"/>
</xsd:choice>
</xsd:sequence>
</xsd:complexType>
<!-- *************** SystemHeader declaration **************************** -->
<xsd:element name="SystemHeader" bs2:ifNext="000001BB"/>
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="SHSC" type="xsd:int"/>
    <xsd:element name="SHLength" type="xsd:unsignedShort"/>
    <xsd:element name="SHFields">
      <xsd:simpleType>
        <xsd:restriction base="bs1:byteRange">
          <xsd:annotation>
            <xsd:appinfo>
              <bs2:length value="../mpg2:SHLength"/>
            </xsd:appinfo>
          </xsd:annotation>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<!-- *************** Stuffing Type ************************--- -->
<!-- !-- <xsd:simpleType name="StuffingType" bs2:if="mpg2:StuffingLength != 0">-->
<xsd:complexType>
  <xsd:restriction base="bs1:byteRange">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:length value="../mpg2:StuffingLength"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>
<!-- !-- >
<!-- *************** Marker Bit ************************--- -->
<xsd:element name="Mkr" type="bs1:b1"/>
<!-- ************ Packet Base Type declaration ******************>
<xsd:complexType name="PacketBaseType" abstract="true">
  <xsd:sequence>
    <xsd:element name="SCPref" type="bs1:b24" fixed="1"/>
    <xsd:element name="StrID" type="xsd:unsignedByte"/>
    <xsd:element name="PktLen" type="xsd:unsignedShort"/>
  </xsd:sequence>
</xsd:complexType>

<!-- ************ Misc Stream ************>
<xsd:complexType name="MiscPacket">
  <xsd:complexContent>
    <xsd:extension base="mpg2:PacketBaseType">
      <xsd:sequence>
        <xsd:element name="Bytes"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- ************ PES Declaration ************>
<xsd:complexType name="PESType">
  <xsd:complexContent>
    <xsd:extension base="mpg2:PacketBaseType">
      <xsd:sequence>
        <xsd:element name="Flags1" type="xsd:unsignedByte"/>
        <xsd:element name="PTSFlag" type="bs1:b1"/>
        <xsd:element name="DTSFlag" type="bs1:b1"/>
        <xsd:element name="Flags2" type="bs1:b6"/>
        <xsd:element name="PESHdrLength" type="xsd:unsignedByte"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:annotation>
  <xsd:appinfo>
    <bs2:length value="../mpg2:PktLen – 3 – ../mpg2:PESHdrLength"/>
  </xsd:appinfo>
</xsd:annotation>
## Appendix F

### Listings for generic syntax translation

**Listing F.1: BS Schema for ID3v2**

```xml
<xsd:schema targetNamespace="urn:id3:v2" xmlns:id3="urn:id3:v2"
    xmlns:bs2="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS"
    xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
    xmlns:bs0="urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:uint="urn:UnsignedIntegers"
bs2:rootElement="id3:mp3File">
    <xsd:import namespace="urn:UnsignedIntegers"
schemaLocation="../schemas/FDIS/UnsignedIntegers.xsd"/>
    <xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
schemaLocation="../schemas/FDIS/BSDL-1.xsd"/>
    <xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS"
schemaLocation="../schemas/FDIS/BSDL-2.xsd"/>
    <xsd:include schemaLocation="ID3frames.xsd"/>
  <!-- ----------- Root element declaration ------------------------------------ -->
  <xsd:element name="mp3File">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="id3:ID3" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="mp3Data" type="bs1:byteRange"/>
      </xsd:sequence>
      <xsd:attribute ref="bs1:bitstreamURI"/>
    </xsd:complexType>
  </xsd:element>
  <!-- ----------- Id3 declaration ------------------------------------------- -->
  <xsd:element name="ID3" bs2:ifNext="494433"/>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="header" type="id3:HeaderType"/>
        <xsd:element name="extendedHeader" type="id3:ExtendedHeaderType"
          minOccurs="0" bs2:if="id3:header/id3:extendedHeaderPresent = 1"/>
      </xsd:sequence>
      <xsd:choice maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

---

F-1
<xsd:element name="footer" type="id3:HeaderType" bs2:ifNext="334449"/>
  <!-- ifNext="'3DI'" --><!-->
<xsd:element name="padding" type="id3:paddingType" bs2:ifNext="00"/>
<xsd:element name="userTextFrame" type="id3:UserFrameType"
  bs2:ifNext="54585858"/> <!-- ifNext="'TXXX'" --><!-->
<xsd:element name="textField" type="id3:TextFrameType" bs2:ifNext="54"/>
  <!-- ifNext="'T'" --><!-->
<xsd:element name="userURLFrame" type="id3:UserFrameType"
  bs2:ifNext="57585858"/> <!-- ifNext="'WXXX'" --><!-->
<xsd:element name="urlFrame" type="id3:URLFrameType" bs2:ifNext="57"/>
  <!-- ifNext="'W'" --><!-->
<xsd:element name="attachedPictureFrame" type="id3:PictureFrameType"
  bs2:ifNext="41504943"/> <!-- ifNext="'APIC'" --><!-->
<xsd:element name="commentFrame" type="id3:CommentFrameType"
  bs2:ifNext="434F4D4D"/> <!-- ifNext="'COMM'" --><!-->
<xsd:element name="fileIdentifierFrame" type="id3:FileIdentifierFrameType"
  bs2:ifNext="55464944"/> <!-- ifNext="'UFID'" --><!-->
<xsd:element name="frame" type="id3:MiscFrameType"/>
</xsd:choice>
</xsd:sequence>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:complexType name="HeaderType">
  <xsd:sequence>
    <xsd:element name="id" type="id3:string3"/>
    <xsd:element name="ver" type="id3:hex2"/>
    <xsd:element name="unsynchronization" type="uint:b1"/>
    <xsd:element name="extendedHeaderPresent" type="uint:b1"/>
    <xsd:element name="experimental" type="uint:b1"/>
    <xsd:element name="footerPresent" type="uint:b1"/>
    <xsd:element name="unusedFlags" type="uint:b4"/>
    <xsd:element name="size" type="id3:synchsafeInt"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ExtendedHeaderType">
  <xsd:sequence>
    <xsd:element name="size" type="id3:synchsafeInt"/>
    <xsd:element name="payload" type="id3:payloadType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="paddingType">
  <xsd:restriction base="bs1:byteRange">
    <xsd:annotation>
      <xsd:appinfo><xsd:appinfo>
        <bs2:startCode value="01 FF"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
Listing F.2: BS Schema for MPEG Multimedia Application Format

```xml
<xsd:schema targetNamespace="urn:edu.uow:mpeg:BSDL:MAF"
 xmlns:maf="urn:edu.uow:mpeg:BSDL:MAF"
 xmlns:bs0="urn:mpeg:mpeg21:2003:01−DIA−BSDL0−NS"
 xmlns:bs1="urn:mpeg:mpeg21:2003:01−DIA−BSDL1−NS"
 xmlns:bs2="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
 bs2:rootElement="maf:MAF"
 xmlns:uint="urn:UnsignedIntegers">
 <xsd:import namespace="urn:UnsignedIntegers" schemaLocation="../schemas/FDIS/UnsignedIntegers.xsd"/>
 <xsd:import namespace="urn:mpeg:mpeg21:2003:01−DIA−BSDL1−NS" schemaLocation="../schemas/FDIS/BSDL−1.xsd"/>
 <xsd:import namespace="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS" schemaLocation="../schemas/FDIS/BSDL−2.xsd"/>
 <xsd:element name="MAF">
 <xsd:complexType>
 <xsd:choice maxOccurs="unbounded">
 <xsd:element name="ftyp" type="maf:FileTypeType" bs0:lookAhead="4" bs0:ifNextStr="ftyp"/>
 <xsd:element name="moov" type="maf:MovieType" bs0:lookAhead="4" bs0:ifNextStr="moov"/>
 <xsd:element name="meta" type="maf:MetadataType" bs0:lookAhead="4" bs0:ifNextStr="meta"/>
 <xsd:element name="mdat" type="maf:MediaDataType" bs0:lookAhead="4" bs0:ifNextStr="mdat"/>
 <xsd:element name="box" type="maf:MiscBoxType"/>
 </xsd:choice>
 </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="BoxType" abstract="true">
 <xsd:sequence>
 <xsd:element name="size" type="xsd:unsignedInt"/>
 <xsd:element name="type" type="maf:FOURCC"/>
 </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="FullBoxType" abstract="true">
 <xsd:complexContent>
 <xsd:extension base="maf:BoxType">
 <xsd:sequence>
 <xsd:element name="version" type="xsd:unsignedByte"/>
 <xsd:element name="flags" type="maf:hex3"/>
 </xsd:sequence>
 </xsd:extension>
 </xsd:complexContent>
 </xsd:complexType>

 <xsd:complexType name="FileTypeType">
 <xsd:complexContent>
 </xsd:complexContent>
 </xsd:complexType>
```

<xsd:extension base="maf:BoxType">
  <xsd:sequence>
    <xsd:element name="major_brand" type="maf:FOURCC"/>
    <xsd:element name="minor_version" type="xsd:unsignedInt"/>
    <xsd:element name="compatible_brand" type="maf:FOURCC" minOccurs="0" maxOccurs="unbounded" bs2:nOccurrences=(maf:size − 16) div 4"/>
  </xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="MovieType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:choice maxOccurs="unbounded" bs0:layerLength="maf:size − 8">
        <xsd:element name="mvhd" type="maf:MiscBoxType" maxOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="mvhd"/>
        <xsd:element name="trak" type="maf:trackType" maxOccurs="unbounded" bs0:lookAhead="4" bs0:ifNextStr="trak"/>
        <xsd:element name="udta" type="maf:MiscBoxType" maxOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="udta"/>
        <xsd:element name="mvex" type="maf:MiscBoxType" maxOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="mvex"/>
        <xsd:element name="meta" type="maf:MetadataType" bs0:lookAhead="4" bs0:ifNextStr="meta"/>
        <xsd:element name="box" type="maf:MiscBoxType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="trackType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:choice maxOccurs="unbounded" bs0:layerLength="maf:size − 8">
        <xsd:element name="tkhd" type="maf:MiscBoxType" minOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="tkhd"/>
        <xsd:element name="tref" type="maf:MiscBoxType" maxOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="tref"/>
        <xsd:element name="edts" type="maf:MiscBoxType" minOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="edts"/>
        <xsd:element name="mdia" type="maf:MediaType" minOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="mdia"/>
        <xsd:element name="udta" type="maf:MiscBoxType" maxOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="udta"/>
        <xsd:element name="meta" type="maf:MetadataType" bs0:lookAhead="4" bs0:ifNextStr="meta"/>
        <xsd:element name="box" type="maf:MiscBoxType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="MetadataType">
<xsd:complexType name="MediaType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:choice maxOccurs="unbounded" bs0:layerLength="maf:size − 8">
        <xsd:element name="mdhd" type="maf:MediaHeaderType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="mdhd"/>
        <xsd:element name="hdlr" type="maf:HandlerType" minOccurs="1" bs0:lookAhead="4" bs0:ifNextStr="hdlr"/>
        <xsd:element name="minf" type="maf:MediaInformationType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="minf"/>
        <xsd:element name="ipmc" type="maf:MiscBoxType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="ipmc"/>
        <xsd:element name="xml" type="maf:XMLType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="xml"/>
        <xsd:element name="other_boxes" type="maf:MiscBoxType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="MediaHeaderType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:choice bs2:if="maf:version = 1">
          <xsd:element name="creation_time1" type="xsd:unsignedLong"/>
          <xsd:element name="modification_time1" type="xsd:unsignedLong"/>
          <xsd:element name="timescale1" type="xsd:unsignedInt"/>
          <xsd:element name="duration1" type="xsd:unsignedLong"/>
        </xsd:choice>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="VideoMediaHeaderType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="graphicsmode" type="xsd:unsignedShort"/>
        <xsd:element name="opcolorRed" type="xsd:unsignedShort"/>
        <xsd:element name="opcolorGreen" type="xsd:unsignedShort"/>
        <xsd:element name="opcolorBlue" type="xsd:unsignedShort"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="SoundMediaHeaderType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="balance" type="xsd:unsignedShort"/>
        <xsd:element name="reserved" type="xsd:unsignedShort"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="HintMediaHeaderType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="maxPDUsize" type="xsd:unsignedShort"/>
        <xsd:element name="avgPDUsize" type="xsd:unsignedShort"/>
        <xsd:element name="maxbitrate" type="xsd:unsignedInt"/>
        <xsd:element name="avgbitrate" type="xsd:unsignedInt"/>
        <xsd:element name="reserved" type="xsd:unsignedInt"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="NullMediaHeaderType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType"/>
    <xsd:extension/>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="PrimaryResourceType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="item_ID" type="xsd:unsignedShort"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="ItemLocationType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="offset_size" type="uint:b4"/>
        <xsd:element name="length_size" type="uint:b4"/>
        <xsd:element name="base_offset_size" type="uint:b4"/>
        <xsd:element name="reserved" type="uint:b4"/>
        <xsd:element name="item_count" type="xsd:unsignedShort"/>
        <xsd:element name="item" bs2:nOccurs="maf:item_count">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="item_ID" type="xsd:unsignedShort"/>
              <xsd:element name="data_reference_index" type="xsd:unsignedShort"/>
              <xsd:element name="base_offset" type="xsd:unsignedInt"/>
              <xsd:element name="extent_count" type="xsd:unsignedShort"/>
              <xsd:element name="extent" bs2:nOccurs="maf:extent_count">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="extent_offset" type="xsd:unsignedInt"/>
                    <xsd:element name="extent_length" type="xsd:unsignedInt"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="baseOffsetTable">  
  <xsd:restriction base="xsd:hexBinary">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:length value="/..maf:base_offset_size"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="extentOffsetTable">  
  <xsd:restriction base="xsd:hexBinary">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:length value="/..maf:offset_size"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="extentLengthType">
  <xsd:restriction base="xsd:hexBinary">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:length value="../../../maf:length_size"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ItemInfoType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="entry_count" type="xsd:unsignedShort"/>
        <xsd:element name="itemInfoEntry" type="maf:ItemInfoEntryType" bs2:nOccurs="maf:entry_count"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ItemInfoEntryType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="item_ID" type="xsd:unsignedShort"/>
        <xsd:element name="item_protection_index" type="xsd:unsignedShort"/>
        <xsd:element name="item_name" type="bs0:stringUTF8NT"/>
        <xsd:element name="content_type" type="bs0:stringUTF8NT" minOccurs="0" bs2:if="maf:size - 16 - string-length(maf:item_name) - 1 - string-length(maf:content_type) - 1 > 0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DataInformationType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:choice maxOccurs="unbounded" bs0:layerLength="maf:size - 8">
        <xsd:element name="url" type="maf:DataEntryUrlType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="url"/>
        <xsd:element name="urn" type="maf:DataEntryUrnType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="urn"/>
        <xsd:element name="dataReference" type="maf:DataReferenceType" minOccurs="0" bs0:lookAhead="4" bs0:ifNextStr="dref"/>  
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DataEntryUrlType">
<xsd:complexType name="DataEntryUrnType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="name" type="bs0:stringUTF8NT"/>
        <xsd:element name="location" type="bs0:stringUTF8NT"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DataReferenceType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="entry_count" type="xsd:unsignedInt"/>
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="url" type="maf:DataEntryUrlType" minOccurs="0"/>
          <xsd:element name="urn" type="maf:DataEntryUrnType" minOccurs="0"/>
        </xsd:choice>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ItemProtectionType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="protection_count" type="xsd:unsignedShort"/>
        <xsd:element name="protection_information" type="maf:MiscBoxType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="XMLType">
  <xsd:complexContent>
    <xsd:extension base="maf:FullBoxType">
      <xsd:sequence>
        <xsd:element name="data" type="bs0:pcdata" layerLength="maf:size - 12"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="MediaDataType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="moov" type="maf:MovieType" bs0:lookAhead="4" bs0:ifNextStr="moov"/>
        <xsd:element name="mdat" type="maf:MediaData" bs0:lookAhead="4" bs0:ifNextStr="mdat"/>
        <xsd:element name="data" type="maf:payloadType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="MiscBoxType">
  <xsd:complexContent>
    <xsd:extension base="maf:BoxType">
      <xsd:sequence>
        <xsd:element name="payload" type="maf:payloadType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="payloadType">
  <xsd:restriction base="bs1:byteRange">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:length value="../maf:size - 8"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="FOURCC">
  <xsd:restriction base="bs0:stringUTF8">
    <xsd:length value="4"/>
    !-- <xsd:annotation>
      <xsd:appinfo>
        <bs0:escape value="true"/>
      </xsd:appinfo>
    </xsd:annotation> -->
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="string32">
  <xsd:restriction base="xsd:normalizedString">
    <xsd:length value="32"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="hex6">
  <xsd:restriction base="xsd:hexBinary">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:schema>
  
  <xsd:simpleType name="hex4">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="4"/>
    </xsd:restriction>
  </xsd:simpleType>
  
  <xsd:simpleType name="hex12">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="12"/>
    </xsd:restriction>
  </xsd:simpleType>
  
  <xsd:simpleType name="hex3">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="3"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
Listing F.3: BBL description of generic syntax translation of ID3v2 metadata into MAF format

<!DOCTYPE bbl [
/maf:meta/maf:xml/maf:data/mp7:Mpeg7/mp7:Description">
<!ENTITY MPEG7PERFORMER "mp7:CreationInformation/mp7:Creation 
/mp7:Creator[mp7:Role/@href='urn:mpeg:mpeg7:RoleCS:2001:PERFORMER'] 
/mp7:Agent/mp7:Name">
]
<!-- **************************************************************************
** BBL Binding
** ===============
**
** Source: Music Player MAF file (.mpa extension)
** Output: MP3 file with ID3v2 tag
**
** Required input variables
** =========================
** $inputFile – a URL locating the MAF file.
** $outputfile – a URL locating the output file.
**
**
** Author: Joseph Thomas – Kerr (jak09@uow.edu.au)
** Date: 1/1/2006
** Revision: 1.0
**
**************************************************************************-->
xmlns:mp7="urn:mpeg:mpeg7:schema:2001"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:maf="urn:edu.uow:mpeg:BSDL:MAF"
xmlns:app="urn:edu.uow:BBL:AppenderParameters"
xmlns:did="urn:mpeg:mpeg21:2002:02–DIDL–NS">
<binding>
<register>
<handler id="defaultHandler" type="Appender">
  <app:outFile>$outputFile</app:outFile>
  <app:noPacketNum>true</app:noPacketNum>
</handler>
<encoder id="deformatMP3" type="MP3onMP4Deformatter"/>
<bbl prefix="maf" bsSchemaLocation="MAF.xsd" fromBinarySource="true"/>
<bbl prefix="id3" bsSchemaLocation=".../id3/id3.xsd"/>
</register>
<variables>
<!-- Lookup mp3 location -->
<define bbl:name="mp3URL" bbl:value="/maf:MAF/maf:meta/maf:xml/maf:data/did:DIDL 
did:Item/did:Item/did:Component/did:Resource[@mimeType='audio/mpeg']/@ref"/>
<define bbl:name="mp3id" bbl:value="/maf:MAF/maf:meta/maf:iinf/maf:itemInfoEntry [maf:item_name=$mp3URL]/maf:item_ID"/>

<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−*
BBL Binding
−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−

Source: Music Player MAF file (.mpa extension)
Output: MP3 file with ID3v2 tag

Required input variables

Author: Joseph Thomas – Kerr (jak09@uow.edu.au)
Date: 1/1/2006
Revision: 1.0

−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−
Listings for generic syntax translation


<!−− Lookup jpeg location −−>


</variables>

</packet>

<content bbl:encoding="deformatMP3">

<id3:mp3File xmlns:bs1="urn:mpeg:mpeg21:2003−DIA−BSDL1−NS" xmlns:id3="urn:id3:v2">
<attribute name="bs1:bitstreamURI">
<value >of select="$inputFile"/></attribute>

<id3:id3>
<id3:id IBID</id3:id>
<id3:ver 0300</id3:ver>
<id3:unsynchronization >0</id3:unsynchronization >
<id3:extendedHeaderPresent >0</id3:extendedHeaderPresent >
<id3:experimental >0</id3:experimental >
<id3:footerPresent >0</id3:footerPresent >
<id3:unusedFlags >0</id3:unusedFlags >
<id3:size >
<id3:byte0 >
<value >of select="subsequence($id3SizeBytes,1,1)"/></id3:byte0 >
<id3:byte1 >
<value >of select="subsequence($id3SizeBytes,2,1)"/></id3:byte1 >
<id3:byte2 >
<value >of select="subsequence($id3SizeBytes,3,1)"/></id3:byte2 >
<id3:byte3 >
<value >of select="subsequence($id3SizeBytes,4,1)"/></id3:byte3 >
</id3:size>
</id3:id3>
</id3:mp3File>
<id3:textFrame>
  <id3:id>TIT2</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3SongTitleLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
  <id3:value>
    <![CDATA[include ref="&MPEG7PATH;/mp7:CreationInformation/mp7:Creation/mp7:Title[@type='songTitle']/text()"/]]>
  </id3:value>
</id3:textFrame>

<id3:textFrame>
  <id3:id>TALB</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3AlbumTitleLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
  <id3:value>
    <![CDATA[include ref="&MPEG7PATH;/mp7:CreationInformation/mp7:Creation/mp7:Title[@type='albumTitle']/text()"/]]>
  </id3:value>
</id3:textFrame>

<id3:textFrame>
  <id3:id>TPE1</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3ArtistLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
  <id3:value>
    <![CDATA[include ref="concat(&MPEG7PATH;/&MPEG7PERFORMER;/mp7:GivenName, ', &MPEG7PATH;/&MPEG7PERFORMER;/mp7:FamilyName)"/]]>
  </id3:value>
</id3:textFrame>

<id3:textFrame>
  <id3:id>TCON</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3GenreLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
  <id3:value>
    <![CDATA[include ref="&MPEG7PATH;/mp7:CreationInformation/mp7:Classification/mp7:Genre/mp7:Name/text()"/]]>
  </id3:value>
</id3:textFrame>

<id3:textFrame>
  <id3:id>TYER</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3YearLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
  <id3:value>
    <![CDATA[include ref="&MPEG7PATH;/mp7:CreationInformation/mp7:CreationCoordinates/mp7:Date/mp7:TimePoint/text()"/]]>
  </id3:value>
</id3:textFrame>

<id3:attachedPictureFrame>
  <id3:id>APIC</id3:id>
  <id3:size>
    <value>
      <![CDATA[$id3ApicLength]]></value>
    </id3:size>
  <id3:flags>
    0000</id3:flags>
  <id3:encoding>
    00</id3:encoding>
</id3:attachedPictureFrame>
<id3:mimeType>image/jpg</id3:mimeType>
=id3:pictureType>00</id3:pictureType>
=id3:description/>
=id3:data>
<value>of select="concat($jpegExtent_offset,' ', $jpegExtent_length)"/>
<!-- should have $jpegBase_offset as well -- but the addition converts to scientific notation... -->
</id3:data>
</id3:attachedPictureFrame>
</id3:ID3>
</id3:mp3Data>
<value>of select="concat($mp3Base_offset + $mp3Extent_offset,' ', $mp3Extent_length)"/>
</id3:mp3Data>
</id3:mp3File>
</variables>
<!-- Calculate ID3 field lengths -->
<define bbl:name="id3SongTitleLength" bbl:value="string-length(./id3:mp3File/id3:ID3/id3:textFrame [id3:id='TIT2']/id3:value) + 2"/>
<!-- 2 = encoding byte + null terminator -->
<define bbl:name="id3AlbumTitleLength" bbl:value="string-length(./id3:mp3File/id3:ID3/id3:textFrame [id3:id='TALB']/id3:value) + 2"/>
<define bbl:name="id3ArtistLength" bbl:value="string-length(./id3:mp3File/id3:ID3/id3:textFrame [id3:id='TPE1']/id3:value) + 2"/>
<define bbl:name="id3YearLength" bbl:value="string-length(./id3:mp3File/id3:ID3/id3:textFrame [id3:id='TYER']/id3:value) + 2"/>
<define bbl:name="id3ApicLength" bbl:value="13 + $jpegExtent_length"/>
<!-- Calculate synchSafe Integer -->
<define bbl:name="id3FrameHeaderLength" bbl:value="10"/>
<define bbl:name="id3TotalLength" bbl:value="6 + $id3FrameHeaderLength + $id3SongTitleLength + $id3AlbumTitleLength + $id3ArtistLength + $id3YearLength + $id3GenreLength + $id3ApicLength"/>
<define bbl:name="id3SizeBytes" bbl:value="bbl:synchSafeInteger($id3TotalLength)"/>
</variables>
</binding>
</bbl>
Appendix G

Listings for virtual container assembly

Listing G.1: MPEG File Format Handler Parameters

```xml
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns="http://whisper.elec.uow.edu.au/mpeg/mpeg21/dis/handler/MPEGFF"
  targetNamespace="http://whisper.elec.uow.edu.au/mpeg/mpeg21/dis/handler/MPEGFF">
  <!-- This element shows how the BBL handler attribute shall be populated -->
  <xs:element name="file">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="compatibleBrand" type="fourCC" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="filename" type="xs:anyURI" use="required"/>
        <!-- FIXME: filename should be passed by BBL engine, not in static description. -->
      <xs:attribute name="majorBrand" type="fourCC" use="required"/>
      <xs:attribute name="minorVersion" type="xs:int" use="optional" default="1"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="movie">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="sdp" type="SDP" minOccurs="0"/>
        <xs:element ref="metadata" minOccurs="0"/>
        <xs:element name="track" type="TrackType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="box" type="BoxType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="timeScale" type="xs:unsignedInt" use="required"/>
      <xs:attribute name="rate" type="xs:float" use="optional" default="1"/>
      <xs:attribute name="volume" type="xs:float" use="optional" default="1"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:complexType name="movieLocationType">
  <xs:attribute name="location" type="movieLocationType" use="optional" default="top"/>
</xs:complexType>

<xs:complexType name="TrackType">
  <xs:sequence>
    <xs:choice>
      <xs:sequence>
        <xs:element name="null" type="NullTrackType"/>
        <xs:element name="sampleDescriptor" type="SampleDescriptionType" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="video" type="VideoTrackType"/>
        <xs:element name="videoSampleDescriptor" type="VideoSampleDescriptionType" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="sound" type="SoundTrackType"/>
        <xs:element name="soundSampleDescriptor" type="SoundSampleDescriptionType" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:choice>
    <xs:element name="dataRef" type="DRefType" maxOccurs="unbounded"/>
    <xs:element name="hint" type="HintType" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="metadata" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
  <xs:attribute name="maxChunkDuration" type="xs:float" use="optional" default="0"/>
  <xs:attribute name="maxChunkSize" type="xs:unsignedInt" use="optional" default="0"/>
  <xs:attribute name="layer" type="xs:unsignedInt" use="optional" default="0"/>
  <xs:attribute name="alternateGroup" type="xs:unsignedInt" use="optional" default="0"/>
  <xs:attribute name="size" type="twoFloats" use="optional" default="1 1"/>
  <xs:attribute name="timeScale" type="xs:unsignedInt" use="required"/>
  <xs:attribute name="volume" type="xs:float" use="optional" default="1"/>
  <xs:attribute name="language" type="string3" use="optional" default="eng"/>
</xs:complexType>
<xs:element name="metadata">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="xml" type="XmlType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="item" type="ItemType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="box" type="BoxType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="handler" type="fourCC" use="required"/>
    <xs:attribute name="desc" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="ItemType">
  <xs:annotation>
    <xs:documentation>
      If the item is internal to the file, then id is used and location is prohibited. If it is external, then uri is used to point to it and location is optional. In this case, if location is absent, then the entire file is assumed.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="info" type="ItemInfoType"/>
    <xs:element name="location" type="ItemLocationType" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="optional">! -- IDREF, but external to the fragment passed to the handler -- -->
  <xs:attribute name="uri" type="xs:anyURI" use="optional"/>
  <xs:attribute name="primaryItem" type="xs:boolean" use="optional"/>
</xs:complexType>

<xs:complexType name="ItemInfoType">
  <xs:attribute name="name" type="xs:normalizedString" use="required"/>
  <xs:attribute name="contentType" type="xs:normalizedString" use="required"/>
  <xs:attribute name="contentEncoding" type="xs:normalizedString" use="optional"/>
</xs:complexType>

<xs:complexType name="ItemLocationType">
  <xs:sequence maxOccurs="unbounded">
    <xs:element name="extent">
      <xs:complexType>
        <xs:attribute name="offset" type="xs:unsignedLong" use="required"/>
        <xs:attribute name="length" type="xs:unsignedLong" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="baseOffset" type="xs:unsignedLong" use="required"/>
</xs:complexType>

<xs:complexType name="XmlType">
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="binarized" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>

<xs:complexType name="HintType">
</xs:complexType>

</xs:annotation>
A payload type of \(-1\) indicates that the payload type is to be specified dynamically using SDP.

If dataRef is absent, then the data is within the file.
Listings for virtual container assembly

<!−−<xs:attribute name="dataRef" use="required" type="xs:IDREF"/>−−>  <!−− no obvious use case for this; no obvious syntax to add the rest of the track data either. −−> </xs:complexType>

<xs:complexType name="SoundSampleDescriptionType">
  <xs:complexContent>
    <xs:extension base="SampleDescriptionType">
      <xs:attribute name="sampleRate" type="xs:float" use="required"/>
      <xs:attribute name="channels" type="xs:unsignedShort" use="optional" default="2"/>
      <xs:attribute name="sampleSize" type="xs:unsignedShort" use="optional" default="16"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="VideoSampleDescriptionType">
  <xs:complexContent>
    <xs:extension base="SampleDescriptionType">
      <xs:attribute name="compressorName" type="string31" use="optional"/>
      <xs:attribute name="dimensions" type="twoShorts" use="required"/>
      <xs:attribute name="dpi" type="twoFloats" use="optional" default="72 72"/>
      <xs:attribute name="frameCount" type="xs:unsignedShort" use="optional" default="1"/>
      <xs:attribute name="depth" type="xs:unsignedShort" use="optional" default="24"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="TRefType">
  <xs:attribute name="ref" use="required" type="xs:IDREF"/>
  <xs:attribute name="type" use="optional" type="fourCC" default="csdc"/>
</xs:complexType>

<!−−<xs:complexType name="DRefType"−−>  <!−− If uri is not provided, then the data reference is internal to the file. −−> </xs:complexType>

<xs:complexType name="BoxType">
  <xs:annotation>
    <xs:documentation> This type lets you create arbitrary boxes to allow extensibility. The box is a Box type by default; if either version or flags attributes are present, the box will be a FullBox. A box is made up of other boxes, and/or BSD elements. Desc is an human-readable string describing the field. It is not used inside the handler. </xs:documentation>
  </xs:annotation>
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="box" type="BoxType"/>
    <xs:group ref="BSDElements"/>
  </xs:choice>
</xs:complexType>
<xs:attribute name="type" type="fourCC" use="required"/>
<xs:attribute name="desc" type="xs:string" use="optional"/>
<xs:attribute name="version" type="xs:unsignedByte" use="optional" default="0"/>
<xs:attribute name="flags" type="hex3" use="optional" default="000000"/>
</xs:complexType>
<xs:group name="BSDElements">
  <xs:annotation>
    <xs:documentation>This group is used to instantiate elements which are interpreted as a BSD. They must be from a namespace which is registered as a BSD namespace within the BBL description.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:group>

<xs:simpleType name="zeroToSeven">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:maxExclusive value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="shortList">
  <xs:list itemType="xs:unsignedShort"/>
</xs:simpleType>
<xs:simpleType name="twoShorts">
  <xs:restriction base="shortList">
    <xs:length value="2"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="threeShorts">
  <xs:restriction base="shortList">
    <xs:length value="3"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="floatList">
  <xs:list itemType="xs:float"/>
</xs:simpleType>
<xs:simpleType name="twoFloats">
  <xs:restriction base="floatList">
    <xs:length value="2"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="fourCC">
  <xs:restriction base="xs:string">
    <xs:length value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="string31">
  <xs:restriction base="xs:string">
    <xs:maxLength value="31"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="string3"/>
<xs:restriction base="xs:NCName">
    <xs:length value="3"/>
</xs:restriction>
</xs:simpleType>
<xs:restriction base="xs:normalizedString"/>
</xs:simpleType>
<xs:simpleType name="hex3">
    <xs:restriction base="xs:hexBinary">
        <xs:length value="3"/>
    </xs:restriction>
</xs:simpleType>

<!-- packet level parameters -->
<xs:element name="item">
    <xs:complexType>
        <xs:attribute name="ref" type="xs:NCName"/>
    </xs:complexType>
</xs:element>
<xs:element name="track">
    <xs:complexType>
        <xs:attribute name="ref" type="xs:NCName"<!-- IDREF, but external to the fragment passed to the handler -->>
    </xs:complexType>
</xs:element>
<xs:element name="sample">
    <xs:complexType>
        <xs:attribute name="firstPacket" type="xs:boolean" use="optional" default="false"/>!
        <xs:attribute name="sampleDescriptor" type="xs:NCName" use="required"<!-- IDREF, but external to the fragment passed to the handler -->>
        <xs:attribute name="compositionTimeDelta" type="xs:int" use="optional" default="0"/>!
    </xs:complexType>
</xs:element>
<xs:element name="hint">
    <xs:complexType>
        <xs:annotation>
            The children of this element represent immediate data to be put into the hint track, before or after the main data (according to position).<!xs:documentation>
        </xs:annotation>
        <xs:group ref="BSDElements" minOccurs="0"/>
        <xs:attribute name="marker" type="xs:boolean" use="optional" default="false"/>
        <xs:attribute name="isBframe" type="xs:boolean" use="optional" default="false"/>
    </xs:complexType>
</xs:element>
</xs:annotation>
</xs:complexType>
<xs:group ref="BSDElements" minOccurs="0"/>
<xs:attribute name="marker" type="xs:boolean" use="optional" default="false"/>
<xs:attribute name="isBframe" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>
</xs:element>
</xs:schema>
Appendix H

Listings for H.264/AVC Streaming and delivery

Listing H.1: BS Schema describing H.264/AVC

```xml
<!DOCTYPE schema [  
<!ENTITY P "0"  
<!ENTITY B "1"  
<!ENTITY I "2"  
<!ENTITY SP "3"  
<!ENTITY SI "4"  
<!ENTITY allP "5"  
<!ENTITY allB "6"  
<!ENTITY allI "7"  
<!ENTITY allSP "8"  
<!ENTITY allSI "9"  

<!ENTITY PorSPslice "($avc:sliceType = &P; or $avc:sliceType = &allP; or  
$avc:sliceType = &SP; or $avc:sliceType = &allSP;)"  
<!ENTITY Bslice "($avc:sliceType = &B; or $avc:sliceType = &allB;)"  
<!ENTITY predictedSlice "(&PorSPslice; or &Bslice;)"  

<!ENTITY SPslice "($avc:sliceType = &SP; or $avc:sliceType = &allSP;)"  
<!ENTITY SIslice "($avc:sliceType = &SI; or $avc:sliceType = &allSI;)" ]>  
<xsd:element name="avc:h264"/>

xmlns:com="urn:edu.uow:CommonTypes"  
xmlns:dia="unnecessary"  
xmlns:bs1x="urn:mpeg:mpeg21:2003:01–DIA–BSDL1x–NS"  
xmlns:bs1="urn:mpeg:mpeg21:2003:01–DIA–BSDL1–NS"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
bs2:rootElement="avc:h264"/>
```

H–1
elementFormDefault="qualified"
attributeFormDefault="unqualified"
xmllns:bs0="urn:mpeg:mpeg21:2003:01–DIA–BSDL0–NS">
<xsd:annotation><xsd:appinfo>
<bs2x:xpathScript ref="functions.js"/>
</xsd:appinfo><xsd:annotation>
<xsd:import namespace="urn:edu.uow:CommonTypes"
schemaLocation="../../../content/BSSchemas/CommonTypes.xsd"/>
schemaLocation="../../../BSDL/schemata/BSDL-1.xsd"/>
schemaLocation="../../../BSDL/schemata/BSDL-1x.xsd"/>
schemaLocation="rdo.xsd"/>
<!−−∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−
Root element declaration ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗−−
><xsd:element name="h264">
<!−−<xsd:annotation><xsd:appinfo>
<bs2x:variable name="naluID" value="0" bs0:execute="before"/>
<bs2x:variable name="referencePictureList" value="()" bs0:execute="before"/>
</xsd:appinfo><xsd:annotation>−−>
<xsd:complexType>
<xsd:sequence maxOccurs="unbounded">
<xsd:group ref="ByteStreamNALUnitGroup"/>
</xsd:sequence>
<xsd:attribute ref="bs1:bitstreamURI"/>
</xsd:complexType>
</xsd:element>

<xsd:group name="ByteStreamNALUnitGroup">
<xsd:sequence>
<xsd:element name="startCode" fixed="00000001"/>
<xsd:complexType>
<xsd:simpleContent>
<xsd:extension base="com:hex4">
<xsd:attribute ref="rdo:offset"/>
</xsd:extension>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>
<xsd:choice>
<xsd:element ref="sequenceParameterSet"/>
<xsd:element ref="pictureParameterSet"/>
<xsd:element ref="sliceBPartition"/>
<xsd:element ref="sliceCPartition"/>
<xsd:element ref="slice"/>
<xsd:element ref="otherNALUnit"/>
</xsd:choice>
<!−−<xsd:element ref="NALUnit"−−>−−>−−>−−>−−>−−><−− substitution group −− doesn’t work
<!−− abstract="true"−−>
<xsd:complexType name="NALUnitType" abstract="true">
  <xsd:sequence>
    <xsd:element name="forbidden_zero_bit" type="bs1:b1" fixed="0"/>
    <!-- --><xsd:annotation><xsd:appinfo>
      <bs2x:variable name="naluID" value="$avc:naluID+1"/>
    </xsd:appinfo></xsd:annotation>
    <xsd:element>
      <xsd:element name="nal_ref_idc" type="bs1:b2"/>
      <xsd:element name="nal_unit_type" type="bs1:b5"/>
    </xsd:sequence>
    <!-- --><xsd:attribute ref="rdo:id" bs0:value="$avc:naluID"/>
    <xsd:attribute ref="rdo:dataUnit"/>
    <xsd:attribute ref="rdo:size"/>
  </xsd:complexType>
</xsd:complexType>

<xsd:complexType name="SpsType">
  <xsd:complexContent>
    <xsd:extension base="NALUnitType">
      <xsd:sequence minOccurs="0" bs2:if="avc:profile_idc = 100 or avc:profile_idc = 110 or avc:profile_idc = 122 or avc:profile_idc = 144">
        <xsd:element name="chroma_format_idc" type="bs1:unsignedExpGolomb"/>
        <xsd:element name="residual_colour_transform_flag" type="bs1:b1" bs2:if="avc:chroma_format_idc = 3"/>
        <xsd:element name="bit_depth_luma_minus8" type="bs1:unsignedExpGolomb"/>
        <xsd:element name="bit_depth_chroma_minus8" type="bs1:unsignedExpGolomb"/>
        <xsd:element name="apprime_y_zero_transform_bypass_flag" type="bs1:b1"/>
        <xsd:element name="seq_scaling_matrix_present_flag" type="bs1:b1"/>
        <!-- --><xsd:annotation><xsd:appinfo>
          <bs2x:variable name="sps" bs2x:position="avc:seq_parameter_set_id + 1"/> <!-- -->
          sequences numbered 1 to n <--> </xsd:appinfo></xsd:annotation>
    </xsd:sequence>
    <xsd:element name="log2_max_frame_num_minus4" type="bs1:unsignedExpGolomb"/>
    <xsd:element name="pic_order_cnt_type" type="bs1:unsignedExpGolomb"/>
    <xsd:element name="log2_max_pic_order_cnt_lsb_minus4" type="bs1:unsignedExpGolomb" minOccurs="0" bs2:if="avc:pic_order_cnt_type = 0"/>
    <xsd:element name="delta_pic_order_signed" type="bs1:b1"/>
    <xsd:element name="offset_for_non_ref_pic" type="bs1:unsignedExpGolomb"/>
  </xsd:extension>
</xsd:complexType>
Listings for H.264/AVC Streaming and delivery

<xsd:element name="offset_for_top_to_bottom_field"
type="bs1:unsignedExpGolomb"/>
<xsd:element name="num_ref_frames_in_pic_order_cnt_cycle"
type="bs1:unsignedExpGolomb"/>
<xsd:element name="offset_for_ref_frame" type="bs1:unsignedExpGolomb"
    minOccurs="0" maxOccurs="unbounded"
    bs2:nOccurs="avc:num_ref_frames_in_pic_order_cnt_cycle"/>
</xsd:sequence>
<xsd:element name="num_ref_frames" type="bs1:unsignedExpGolomb"/>
<xsd:element name="gaps_in_frame_num_value_allowed_flag" type="bs1:b1"/>
<xsd:element name="pic_width_in_mbs_minus1"
type="bs1:unsignedExpGolomb"/>
<xsd:element name="pic_height_in_map_units_minus1"
type="bs1:unsignedExpGolomb"/>
<xsd:element name="frame_mbs_only_flag" type="bs1:b1"/>
<xsd:element name="mb_adaptive_frame_field_flag" type="bs1:b1"
    bs2:if="avc:frame_mbs_only_flag = 1" minOccurs="0"/>
<xsd:element name="direct_8x8_inference_flag" type="bs1:b1"/>
<xsd:sequence minOccurs="0" bs2:if="avc:frame_cropping_flag = 1"/>
<xsd:element name="frame_cropping_flag" type="bs1:b1"/>
</xsd:sequence>
<xsd:group ref="rbspTrailingBits"/>
<xsd:element name="restOfNAL" type="restOfNAL"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>

<xsd:complexType name="PPSType" bs2:ifNextMask="1F"
    bs2:ifNext="08"><!-- substitutionGroup="avcNALUnit" -->
<xsd:annotation>
    <xsd:appinfo>
        <bs2x:variable name="pps" bs2x:position="avc:pic_parameter_set_id + 1"/>
    </xsd:appinfo>
</xsd:annotation>
</xsd:element>

<xsd:complexType name="PPSType">
</xsd:complexType>
</xsd:contentType>
<xsd:element name="entropy_coding_mode_flag" type="bs1:b1"/>
<xsd:element name="pic_order_present_flag" type="bs1:b1"/>
<xsd:element name="num_slice_groups_minus1" type="bs1:unsignedExpGolomb"/>
<xsd:group ref="sliceGroups" minOccurs="0">
  bs2:if="avc:num_slice_groups_minus1 > 0"/>
<xsd:element name="num_ref_idx_l0_active_minus1" type="bs1:unsignedExpGolomb"/>
<xsd:element name="num_ref_idx_l1_active_minus1" type="bs1:unsignedExpGolomb"/>
<xsd:element name="weighted_pred_flag" type="bs1:b1"/>
<xsd:element name="weighted_bipred_idc" type="bs1:b2"/>
<xsd:element name="pic_init_qp_minus26" type="bs1:signedExpGolomb"/>
<xsd:element name="pic_init_qs_minus26" type="bs1:signedExpGolomb"/>
<xsd:element name="chroma_qp_index_offset" type="bs1:signedExpGolomb"/>
<xsd:element name="constrained_intra_pred_flag" type="bs1:b1"/>
<xsd:element name="deblocking_filter_control_present_flag" type="bs1:b1"/>
<xsd:group ref="ppsExtension" minOccurs="0" bs2:ifNext="00 7F"/>
<!−− if( more_rbsp_data( ) ) −−>
<xsd:group ref="rbspTrailingBits"/>
</xsd:sequence>

<xsd:element name="slice_group_map_type" type="bs1:unsignedExpGolomb"/>
<xsd:choice>
  <xsd:sequence bs2:if="slice_group_map_type = 0">
    <xsd:element name="run_length_minus1" type="bs1:unsignedExpGolomb"/>
  </xsd:sequence>
  <xsd:sequence bs2:if="slice_group_map_type = 2">
    <xsd:sequence bs2:nOccurs="avc:num_slice_groups_minus1 + 1">
      <xsd:element name="top_left" type="bs1:unsignedExpGolomb"/>
      <xsd:element name="top_right" type="bs1:unsignedExpGolomb"/>
    </xsd:sequence>
  </xsd:sequence>
  <xsd:sequence bs2:if="slice_group_map_type = 3 or slice_group_map_type = 4 or slice_group_map_type = 5">
    <xsd:element name="slice_group_change_direction_flag" type="bs1:b1"/>
    <xsd:element name="slice_group_change_rate_minus1" type="bs1:unsignedExpGolomb"/>
  </xsd:sequence>
  <xsd:sequence bs2:if="slice_group_map_type = 6">
    <xsd:element name="pic_size_in_map_units_minus1" type="bs1:unsignedExpGolomb"/>
    <xsd:element name="slice_group_id" type="xsd:unsignedInt"/>
    <xsd:sequence bs2:nOccurs="avc:pic_size_in_map_units_minus1 + 1"/>
  </xsd:sequence>
</xsd:choice>
</xsd:sequence>

<!−− TODO should have dynamic size−−>
<xsd:choice>
  <xsd:sequence>
    <xsd:group name="ppsExtension">
      <xsd:sequence>
        <xsd:element name="transform_8x8_mode_flag" type="bs1:b1" />
        <xsd:element name="pic_scaling_matrix_present_flag" type="bs1:b1" />
        <xsd:sequence bs2:if="avc:pic_scaling_matrix_present_flag = 1">
          <xsd:element name="pic_scaling_list" bs2:nOccurs="6 + 2 * avc:transform_8x8_mode_flag"/>
        </xsd:sequence>
      </xsd:sequence>
    </xsd:group>
    <xsd:element name="present_flag" type="bs1:b1" />
    ![if (i < 6 )
      scaling_list( ScalingList4x4[ i ], 16,
        UseDefaultScalingMatrix4x4Flag[ i ] )
    else
      scaling_list( ScalingList8x8[ i6 ], 64,
        UseDefaultScalingMatrix8x8Flag[ i6 ] )
    ] -->
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="second_chroma_qp_index_offset" type="bs1:signedExpGolomb" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="slice" type="sliceNALunit" bs2:ifNextMask="1F" bs2:ifNext="01 05">
  ![-- substitutionGroup="avc:NALUnit" -->
  <xsd:annotation><xsd:appinfo>
    ![if (rdo:newPicture(avc:nal_ref_idc, avc:nal_unit_type,
      avc:pic.parameter_set_id,
      avc:frame_num, avc:pic_order_cnt_lsb) and
      avc:nal_ref_idc != 0) then if (count($avc:referencePictureList) = $avc:sliceSPS/avc:num_ref_frames)
        then
          (subsequence($avc:referencePictureList,2),xsd:string($avc:naluID))
        else ($avc:referencePictureList,xsd:string($avc:naluID))
      else $avc:referencePictureList"
    then
    reordering? -->
  </xsd:appinfo></xsd:annotation>
</xsd:element>

<xsd:complexType name="sliceNALunit"/>
<xsd:complexType name="NALUnitType"/>
<xsd:extension base="NALUnitType">
  <xsd:extension base="NALUnitType">
    <xsd:sequence>
      <xsd:group ref="SliceHeaderType"/>
      <xsd:element name="restOfNAL" type="restOfNAL"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>
or =0 -->
</xsd:sequence>
<xsd:element minOccurs="0" bs2:if="$avc:slicePPS/avc:redundant_pic_cnt_present_flag = 1" name="redundant_pic_cnt" type="bs1:unsignedExpGolomb" />
<xsd:element minOccurs="0" bs2:if="&Bslice;" name="direct_spatial_mv_pred_flag" type="bs1:b1" />
<xsd:sequence minOccurs="0" bs2:if="&Bslice; or &SPslice;">
<xsd:element minOccurs="0" name="num_ref_idx_active_override_flag" type="bs1:b1" />
<xsd:sequence minOccurs="0" bs2:if="avc:num_ref_idx_active_override_flag = 1" name="num_ref_idx_l0_active_minus1" type="bs1:unsignedExpGolomb" />
<xsd:element minOccurs="0" bs2:if="&Bslice;" name="num_ref_idx_l1_active_minus1" type="bs1:unsignedExpGolomb" />
</xsd:sequence>
</xsd:element>
<xsd:element minOccurs="0" bs2:if="&PorSPslice; or &Bslice;" name="pred_weight_table" type="PredictionWeightTableType" />
<xsd:element minOccurs="0" bs2:if="avc:nal_ref_idc != 0" name="dec_ref_pic_marking" type="decRefPicMarkingType" />
<xsd:element minOccurs="0" bs2:if="$avc:slicePPS/avc:entropy_coding_mode_flag = 1 and &predictedSlice;" name="cabac_init_idc" type="bs1:unsignedExpGolomb" />
<xsd:element minOccurs="0" bs2:if="&SPslice; or &SIslice;" name="slice_qscale" type="bs1:signedExpGolomb" />
</xsd:element>
</xsd:sequence>
<xsd:element minOccurs="0" bs2:if="$avc:slicePPS/avc:slice_group_map_type > 2 and $avc:slicePPS/avc:slice_group_map_type < 6" name="slice_group_change_cycle" type="SliceGroupChangeCycleType" />
</xsd:element>
<xsd:sequence>
</xsd:group>
</xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:group>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="&predictedSlice;">
  <xsd:element name="ref_pic_list_reordering_flag_l0" type="bs1:b1"/>
  <xsd:annotation>
    <bs2x:variable name="incomplete" value="true()"/>
  </xsd:annotation>
</xsd:element>
<xsd:element name="RefPicReorderingList0" type="PicNumReorderingLoop"
  minOccurs="0" bs2:if="avc:ref_pic_list_reordering_flag_l0 = 1 and $avc:incomplete" maxOccurs="unbounded"/>
<xsd:sequence bs2:if="&Bslice;" minOccurs="0">
  <xsd:element name="ref_pic_list_reordering_flag_l1" type="bs1:b1"/>
  <xsd:annotation>
    <bs2x:variable name="incomplete" value="true()"/>
  </xsd:annotation>
</xsd:element>
<xsd:element name="RefPicReorderingList1" type="PicNumReorderingLoop"
  minOccurs="0" bs2:if="avc:ref_pic_list_reordering_flag_l1 = 1 and $avc:incomplete" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:sequence>

<xsd:complexType name="PicNumReorderingLoop">
  <xsd:sequence>
    <xsd:element name="reordering_of_pic_nums_idc" type="bs1:unsignedExpGolomb"/>
    <xsd:annotation>
      <bs2x:variable name="incomplete" value=". != 3"/>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="abs_diff_pic_num_minus1" type="bs1:unsignedExpGolomb"
    minOccurs="0" bs2:if="avc:reordering_of_pic_nums_idc[last()] = 0 or avc:reordering_of_pic_nums_idc[last()] = 1"/>
  <xsd:element name="long_term_pic_num" type="bs1:unsignedExpGolomb"
    minOccurs="0" bs2:if="avc:reordering_of_pic_nums_idc[last()] = 2"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="sliceBPartition" type="OtherNALUnitType" bs2:ifNextMask="1F" bs2:ifNext="03"/>
<xsd:element name="sliceCPartition" type="OtherNALUnitType" bs2:ifNextMask="1F" bs2:ifNext="04"/>
<xsd:element name="otherNALUnit" type="OtherNALUnitType"/>
</xsd:complexType>

<xsd:extension base="NALUnitType">
  <xsd:sequence>
    <xsd:element name="rbsp" type="rbspType"/>
  </xsd:sequence>
</xsd:extension>
<xsd:complexType name="rbspType">  
  <xsd:restriction base="bs1:byteRange">  
    <xsd:annotation>  
      <xs:appinfo>  
        <bs2:startCode value="00000001"/>  
      </xs:appinfo>  
    </xsd:annotation>  
  </xsd:restriction>  
</xsd:complexType>  

<xsd:simpleType name="frameNumType">  
  <xsd:restriction base="xsd:unsignedLong">  
    <xsd:annotation>  
      <xs:appinfo>  
        <bs2:bitLength value="$avc:sliceSPS/avc:log2_max_frame_num_minus4 + 4"/>  
      </xs:appinfo>  
    </xsd:annotation>  
  </xsd:restriction>  
</xsd:simpleType>  

<xsd:simpleType name="picOrderCntType">  
  <xsd:restriction base="xsd:unsignedLong">  
    <xsd:annotation>  
      <xs:appinfo>  
        <bs2:bitLength value="$avc:sliceSPS/avc:log2_max_pic_order_cnt_lsb_minus4 + 4"/>  
      </xs:appinfo>  
    </xsd:annotation>  
  </xsd:restriction>  
</xsd:simpleType>  

<xsd:schema>  
</xsd:schema>
Listing H.2: BBL binding describing H.264/AVC content delivery over RTP

<!-- ----------------------------------------------------------
** BBL Binding
** ===========
**
** Source: H.264/AVC elementary stream (.264 or .h264 extension)
** Output: RTP stream compliant to RFC3984 (single NAL unit mode)
**
** Required input variables
** =========================
** $framesPerPeriod -- the number of frames to be transmitted
** in a single period.
** $framePeriod -- the length of a frame period in milliseconds.
**
**
** Author: Joseph Thomas - Kerr (jak09@uow.edu.au)
** Date: 19/12/2005
** Revision: 1.0
**
*---------------------------------------------------------->

<binding xmlns="urn:mpeg:mpeg21:2006:01 - BBL - NS"
xmlns:bbl="urn:mpeg:mpeg21:2006:01 - BBL - NS"
xmlns:app="urn:edu.uow:BBL:RTPdumpParameters"
xmlns:avc="urn:H.264/AVC:ES"
xsi:schemaLocation="urn:mpeg:mpeg21:2006:01 - BBL - NS ../Schemas/BBL20.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <variables>
    <define bbl:name="frameTime" bbl:value="($framePeriod div 1000) div $framesPerPeriod"/>
    <define bbl:name="notFirstNAL" bbl:value="false()"/>
    <define bbl:name="newAU" bbl:value="false()"/>
    <define bbl:name="lastSlice" bbl:value="/"/>
    <define bbl:name="delTime" bbl:value="0"/>
    <define bbl:name="expectedPicOrder" bbl:value="0"/>
    <define bbl:name="timestampOffset" bbl:value="0"/>
    <define bbl:name="lastRefIDC" bbl:value="0"/>
    <define bbl:name="refIDC" bbl:value="0"/>
    <define bbl:name="picOrder" bbl:value="0"/>
  </variables>

  <register>
    <handler id="default" type="RTPdump">
      <app:outFile>$outFile</app:outFile>
      <app:logFile>output/carphone.txt</app:logFile>
      <app:noPacketNum>true</app:noPacketNum>
      <app:frameRate>?1 div $frameTime?</app:frameRate>
    </handler>
  </register>

  <packetStream>
    <params>
      <app:timestampOffset>$timestampOffset</app:timestampOffset>
      <app:newAU>$newAU</app:newAU>
    </params>
  </packetStream>
</binding>
Listings for H.264/AVC Streaming and delivery

<params>
  <!-- test output -->
  <contentTemplate>
    <include ref="/avc:h264/\*[name()!='startCode']" depth="-1" ancestors="-1"/>
    <!-- sending pps & sps in band to match h264 rtpdump output -->
    <timing>
      <delTimes bbl:match="." bbl:value="$delTime"/>
    </timing>
    <fragmentation>
      <count value="1" bbl:match="."/>
    </fragmentation>
  </include>
  </contentTemplate>
  <variables>
    <!-- would be more efficient in an XPath function -->
    <assign bbl:name="lastRefIDC" bbl:value="$lastSlice/avc:nal_ref_idc/text()"/>
    <assign bbl:name="refIDC" bbl:value="/avc:h264/avc:slice/avc:nal_ref_idc/text()"/>
    <assign bbl:name="picOrder"
      bbl:value="/avc:h264/avc:slice/avc:pic_order_cnt_lsb/text()"/>
    <assign bbl:name="newAU" bbl:value=""
      $notFirstNAL and (($lastSlice/avc:nal_unit_type/text() &gt;= 1 and
      (($nalType &gt;= 6 and $nalType &lt;= 9) or
      ($nalType &gt;= 14 and $nalType &lt;= 18)))
      or (/avc:h264/avc:slice/avc:frame_num/text() != $lastSlice/avc:frame_num/text())
      or (/avc:h264/avc:slice/avc:pic_parameter_set_id/text() !=
      $lastSlice/avc:pic_parameter_set_id/text())
      or ($refIDC = 0 or $lastRefIDC = 0) and
      ($refIDC != $lastRefIDC)
      or ($picOrder != $lastSlice/avc:pic_order_cnt_lsb/text())
    )
    <assign bbl:name="notFirstNAL" bbl:value="true()"/>
    <assign bbl:name="delTime" bbl:value="if ($newAU) then $delTime + $frameTime
      else $delTime"/>
    <!-- identify whether a timestamp Offset is required. -->
    <assign bbl:name="expectedPicOrder" bbl:value="if ($nalType = 5) then 0
      else if ($newAU) then $expectedPicOrder + 2 else $expectedPicOrder"/>
    <assign bbl:name="timestampOffset" bbl:value="if (/avc:h264/avc:slice) then
      $frameTime + ($picOrder - $expectedPicOrder) div 2
      else $timestampOffset"/>
    <assign bbl:name="lastSlice" bbl:value="/avc:h264/avc:slice else $lastSlice"/>
  </variables>
</packetStream>
Listing H.3: BBL binding describing H.264/AVC content delivery via an ISO file package

```xml
<![CDATA[---************************************************************************---]
** BBL Binding
** ============
**
** Source: H.264/AVC elementary stream (.264 or .h264 extension)
** Output: Hinted MPEG File suitable for RTP streaming
**
** Required input variables
** ==============
** $framesPerPeriod – the number of frames to be transmitted
** in a single period.
** $framePeriod – the length of a frame period in milliseconds.
**
** Author: Joseph Thomas – Kerr (jak09@uow.edu.au)
** Date: 6/3/2006
** Revision: 1.0
**
************************************************************************---]>

<bbl xmlns="urn:mpeg:mpeg21:2006:01-BBL-NS"
     xmlns:bbl="urn:mpeg:mpeg21:2006:01-BBL-NS"
     xmlns:mpff="urn:edu.uow:BBL:MPEGFFParameters"
     xmlns:avc="urn:H.264/AVC:ES"
     xmlns:avcFF="urn:edu.uow:BBL:AVCBoxes"
     xsi:schemaLocation="urn:mpeg:mpeg21:2006:01-BBL-NS ../schemas/BBL22.xsd"
     urn:edu.uow:BBL:MPEGFFParameters ..../schemas/MPEGFFHandlerParameters.xsd
     urn:edu.uow:BBL:AVCBoxes AVCBoxes.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <binding>
        <variables>
            <define bbl:name="sps" bbl:value="/avc:h264/avc:sequenceParameterSet[1]"/>
            <define bbl:name="pps" bbl:value="/avc:h264/avc:pictureParameterSet[1]"/>
            <define bbl:name="frameTicks" bbl:value="$ticksPerSec idiv $framesPerSec"/>
            <define bbl:name="frameTime" bbl:value="1 div $framesPerSec"/>
            <define bbl:name="newAU" bbl:value="true()"/>
            <define bbl:name="lastSlice" bbl:value=""/>
            <define bbl:name="delTime" bbl:value="- $frameTime"/>
            <define bbl:name="picOrderIncrement" bbl:value="2"/>
            <define bbl:name="expectedPicOrder" bbl:value="- $picOrderIncrement"/>
            <define bbl:name="compDelta" bbl:value="0"/>
            <define bbl:name="lastRefIDC" bbl:value="-1"/>
            <define bbl:name="refIDC" bbl:value="0"/>
            <define bbl:name="picOrder" bbl:value="0"/>
        </variables>
        <register>
            <function bbl:name="bbl:size" args="1"/>
            <handler id="default" type="MpegFF">
                <mpff:file majorBrand="isom" filename="?$outFile?"/>
            </handler>
        </register>
    </binding>
</bbl>
```
<mpff:sdp>zzz</mpff:sdp>
<mpff:track id="videoTrack" timeScale="1001">
<mpff:video/>
<!-- Track has a video minf header with the default parameters -->
<mpff:videoSampleDescriptor dimensions="176 144" id="sd" codingName="avc1">
<!-- an AVC sample descriptor -->
<mpff:box type="avcC" desc="config">
<avcFF:AVCDescConfigurationRecord>
<avcFF:ver>1</avcFF:ver>
<avcFF:profile>$sps/avc:profile_idc/text()?</avcFF:profile>
<avcFF:compatibility>$sps/avc:constraint_set_idc/text()?</avcFF:compatibility>
<avcFF:level>$sps/avc:level_idc/text()?</avcFF:level>
<avcFF:reserved>−1</avcFF:reserved>
<avcFF:lengthSizeMinusOne>1</avcFF:lengthSizeMinusOne>
<avcFF:reserved2>−1</avcFF:reserved2>
<avcFF:numSPPS>1</avcFF:numSPPS>
<avcFF:spsLength>9</avcFF:spsLength>
<!-- ?bbl:size($pps)? -->
<avcFF:sps>$sps/avc:slice/text()?</avcFF:sps>
<avcFF:pps>1</avcFF:pps>
<avcFF:ppsLength>5</avcFF:ppsLength>
<!-- ?bbl:size($pps)? -->
<avcFF:pps>$pps/avc:slice/text()?</avcFF:pps>
<avcFF:AVCDescConfigurationRecord>
</mpff:box>
</mpff:videoSampleDescriptor>
<mpff:sdp>
m=video 0 RTP/AVP 96
b=AS:139
a=rtpmap:96 H264/90000
a=fmtp:96 profile—level—id=profile,compat,level in hex;
 sprop—parameter—sets=sps,pps in base64 with comma. Need
 bsdtobin xpath function for this.
</mpff:sdp>
</mpff:hint>
</mpff:track>
</mpff:movie>
</handler>
<bsd prefix="avc" bsSchemaLocation="h264.xsd" fromBinarySource="false"/>
<bsd prefix="avcFF" bsSchemaLocation="AVCBoxes.xsd" fromBinarySource="false"/>
</register>
<packetStream>
<params>
<mpff:track ref="videoTrack"/>
<mpff:sample sampleDescriptor="sd" compositionTimeDelta="?$compDelta?"
 firstPacket="?$newAU?"/>
<mpff:hint isBframe="false" marker="?$newAU?"/>
</params>
<contentTemplate>
<include ref="/avc:h264/avc:slice" depth="−1" ancestors="−1"/>
<timing>
  <delTimes bbl:match="." bbl:value="$delTime"/>
</timing>
</fragmentation>
  <count value="1" bbl:match="."/>
</fragmentation>
</include>
</contentTemplate>

<variables>
<!-- TODO: would be more efficient in an XPath function -->
<assign bbl:name="lastRefIDC" bbl:value="$lastSlice/avc:nal_unit/text()"/>
<assign bbl:name="refIDC" bbl:value="/h264/avc:slice/avc:nal_ref_idc/text()"/>
<assign bbl:name="picOrder"
  bbl:value="/h264/avc:slice/avc:pic_order_cnt_lsb/text()"/>
<assign bbl:name="newAU"
  bbl:value="not($lastRefIDC) or (($lastSlice/avc:nal_unit_type/text() &gt;= 1
    and ($nalType &gt;= 6 and $nalType &lt;= 9)) or
    ($nalType &gt;= 14 and $nalType &lt;= 18)))
    or ($lastSlice/avc:frame_num/text() != $lastSlice/avc:frame_num/text())
    or ($lastSlice/avc:pic_parameter_set_id/text() !=
      $lastSlice/avc:pic_parameter_set_id/text())
    or ($refIDC = 0 or $lastRefIDC = 0) and
    ($refIDC != $lastRefIDC)
    or ($picOrder != $lastSlice/avc:pic_order_cnt_lsb/text())")/>
<assign bbl:name="delTime" bbl:value="if ($newAU) then $delTime + $frameTime
    else $delTime"/>
<!-- identify whether a timestamp Offset is required. -->
<assign bbl:name="expectedPicOrder"
  bbl:value="if ($nalType = 5)
    then 0
    else if ($newAU) then $expectedPicOrder + $picOrderIncrement
    else $expectedPicOrder"/>
<assign bbl:name="compDelta"
  bbl:value="if (/h264/avc:slice)
    then $frameTicks * ($picOrder - $expectedPicOrder + $picOrderIncrement) idiv $picOrderIncrement
    else $compDelta "/>
<!-- +2?? -->
<assign bbl:name="lastSlice"
  bbl:value="if (/h264/avc:slice)
    then ./h264/avc:slice
    else $lastSlice"/>
</variables>
</packetStream>
</bbl>
Appendix I

Listings for Intelligent Media Delivery

Listing I.1: XML Schema for RDO metadata

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://whisper.elec.uow.edu.au/rateDistortionOptimization"
    xmlns:bs0="urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS"
    xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS">
    <xs:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
schemaLocation="../../../BSDL/schemata/BSDL-1.xsd"/>
    <xs:attribute name="dataUnit" type="xs:boolean" default="true"/>
    <xs:attribute name="offset" type="xs:unsignedLong" bs0:value="$bs0:offset"/>
    <xs:attribute name="size" type="xs:unsignedLong"/>
    <xs:attribute name="id" type="xs:unsignedLong"/>
    <xs:attribute name="groupId" type="xs:unsignedLong"/>
    <xs:attribute name="ancestors">
        <xs:simpleType>
            <xs:list itemType="xs:unsignedLong"/>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="duration" type="xs:double"/>
    <xs:attribute name="timestamp" type="xs:double"/>
</xs:schema>
```
Listing I.2: BSDL Schema for MPEG SLS Audio

```xml
<!DOCTYPE schema [
  <!ENTITY ID SCE "0" >
  <!ENTITY ID CPE "1" >
  <!ENTITY ID CCE "2" >
  <!ENTITY ID LFE "3" >
  <!ENTITY ID DSE "4" >
  <!ENTITY ID PCE "5" >
  <!ENTITY ID FIL "6" >
  <!ENTITY ID END "7" >
]>
<xsd:schema targetNamespace="urn:ISO/IEC-14496-3:2007"
  xmlns="urn:ISO/IEC-14496-3:2007"
  xmlns:sls="urn:ISO/IEC-14496-3:2007"
  xmlns:bs2x="urn:mpeg:mpeg21:2003:01-DIA-BSDL2x-NS"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:bs0="urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS"
  xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
  xmlns:bs2="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS"
  xmlns:uow="urn:edu.uow:CommonTypes"
  elementFormDefault="qualified"
  bs2:rootElement="SLS"
  bs2:defaultTreeInMemory="true">
  <xsd:annotation>
    <xsd:appinfo>
      <bs2:parameter name="sampleFrequency">48000</bs2:parameter>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
    schemaLocation="../../../BSDL/schemata/BSDL1.xsd"/>
    schemaLocation="rdo.xsd"/>
  <xsd:element name="SLS">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="lle_individual_channel_stream" type="lle_individual_channel_stream" bs2:nOccurs="$sls:channel_number" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="lle_channel_pair_element">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2x:variable name="timestamp_us" value="if (empty($sls:timestamp_us)) then 0 else $sls:timestamp_us+$sls:duration_us" bs0:execute="after"/>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="lle_individual_channel_stream" type="lle_individual_channel_stream" bs2:nOccurs="$sls:channel_number" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
<xsd:complexType name="lle_individual_channel_stream">
  <xsd:sequence>
    <xsd:element name="lle_ics_length" type="bs1:unsignedShortLE"/>
    <xsd:element name="payload">
      <xsd:simpleType>
        <xsd:restriction base="bs1:byteRange">
          <xsd:annotation>
            <xsd:appinfo>
              <bs2:length value="../sls:lle_ics_length"/>
            </xsd:appinfo>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:complexType>

<xsd:complexType name="SLSSpecificConfig">
  <xsd:sequence>
    <xsd:element name="pcmWordLength" type="bs1:b3"/>
    <xsd:element name="aac_core_present" type="bs1:b1"/>
    <xsd:element name="lle_main_stream" type="bs1:b1"/>
    <xsd:element name="reserved_bit" type="bs1:b1"/>
    <xsd:element name="frameLength" type="bs1:b3"/>
    <xsd:annotation>
      <xsd:appinfo>
        <bs2x:variable name="frameLength" value="xsd:integer(.)"/>
        <bs2x:variable name="samplesPerFrame" value="(1024,2048,4096)[$sls:frameLength+1]"/>
        <bs2x:variable name="duration_us" value="xsd:long($sls:samplesPerFrame div number($sls:sampleFrequency)∗1e6)"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="mpeg4audio">
  <xsd:choice>
    <xsd:element name="Null" bs2:ifNext="0" type="Object"/>
    <xsd:element name="AACmain" bs2:ifNext="1" type="Object"/>
    <xsd:element name="AAC-LC" bs2:ifNext="2" type="Object"/>
    <xsd:element name="AAC-SSR" bs2:ifNext="3" type="Object"/>
    <xsd:element name="AAC-LTP" bs2:ifNext="4" type="Object"/>
    <xsd:element name="SBR" bs2:ifNext="5" type="Object"/>
    <xsd:element name="AAC-Scalable" bs2:ifNext="6" type="Object"/>
    <xsd:element name="TwinVQ" bs2:ifNext="7" type="Object"/>
    <xsd:element name="CELP" bs2:ifNext="8" type="Object"/>
    <xsd:element name="HVXC" bs2:ifNext="9" type="Object"/>
    <xsd:element name="reserved1" bs2:ifNext="10" type="Object"/>
    <xsd:element name="reserved2" bs2:ifNext="11" type="Object"/>
  </xsd:choice>
</xsd:complexType>
<xsd:element name="TTSI" bs2:ifNext="12" type="Object"/>
<xsd:element name="MainSynthetic" bs2:ifNext="13" type="Object"/>
<xsd:element name="WavetableSynthesis" bs2:ifNext="14" type="Object"/>
<xsd:element name="GeneralMIDI" bs2:ifNext="15" type="Object"/>
<xsd:element name="AlgorithmicSynthesisAndAudioFX" bs2:ifNext="16" type="Object"/>
<xsd:element name="ER−AAC−LC" bs2:ifNext="17" type="Object"/>
<xsd:element name="reserved3" bs2:ifNext="18" type="Object"/>
<xsd:element name="ER−AAC−LTP" bs2:ifNext="19" type="Object"/>
<xsd:element name="ER−AAC−scalable" bs2:ifNext="20" type="Object"/>
<xsd:element name="ER−TwinVQ" bs2:ifNext="21" type="Object"/>
<xsd:element name="ER−BSAC" bs2:ifNext="22" type="Object"/>
<xsd:element name="ER−AAC−LD" bs2:ifNext="23" type="Object"/>
<xsd:element name="ER−CELP" bs2:ifNext="24" type="Object"/>
<xsd:element name="ER−HVXC" bs2:ifNext="25" type="Object"/>
<xsd:element name="ER−HILN" bs2:ifNext="26" type="Object"/>
<xsd:element name="ER−Parametric" bs2:ifNext="27" type="Object"/>
<xsd:element name="SSC" bs2:ifNext="28" type="Object"/>
<xsd:element name="PS" bs2:ifNext="29" type="Object"/>
<xsd:element name="MPEGSurround" bs2:ifNext="30" type="Object"/>
<xsd:element name="escape" bs2:ifNext="31" type="Object"/>
<xsd:element name="Layer−1" bs2:ifNext="32" type="Object"/>
<xsd:element name="Layer−2" bs2:ifNext="33" type="Object"/>
<xsd:element name="Layer−3" bs2:ifNext="34" type="Object"/>
<xsd:element name="DST" bs2:ifNext="35" type="Object"/>
<xsd:element name="ALS" bs2:ifNext="36" type="Object"/>
<xsd:element name="SLS" bs2:ifNext="37" type="Object"/>
<xsd:element name="SLSnon−core" bs2:ifNext="38" type="Object"/>
<xsd:element name="reserved4" bs2:ifNext="39" type="Object"/>
<xsd:element name="SMRSimple" bs2:ifNext="40" type="Object"/>
<xsd:element name="SMRMain" bs2:ifNext="41" type="Object"/>
<xsd:element name="reserved5" type="Object"/>
</xsd:choice>
</xsd:complexType>

<xsd:complexType name="Object"></xsd:complexType>

<xsd:group name="adif_sequence" bs2:ifNext=""ADIF"">
  <xsd:sequence>
    <xsd:element ref="adif_header"/>
    <xsd:element ref="byte_alignment"/>
    <xsd:group ref="raw_data_block" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:group>

<xsd:element name="adif_header">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="adif_id" type="FOURCC" fixed="ADIF"/>
      <xsd:element name="copyright_id_present" type="bs1:b1"/>
      <xsd:element name="copyright_id" type="b72" bs2:if="sls:copyright_id_present = 0" minOccurs="0"/>
      <xsd:element name="original_copy" type="bs1:b1"/>
      <xsd:element name="home" type="bs1:b1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:complexType name="ProgramConfigElement">
  <xsd:sequence>
    <xsd:element name="element_instance_tag" type="bs1:b4"/>
    <xsd:element name="object_type" type="bs1:b2"/>
    <xsd:element name="sampling_frequency_index" type="bs1:b4"/>
    <xsd:element name="num_front_channel_elements" type="bs1:b4"/>
    <xsd:element name="num_side_channel_elements" type="bs1:b4"/>
    <xsd:element name="num_back_channel_elements" type="bs1:b4"/>
    <xsd:element name="num_lfe_channel_elements" type="bs1:b2"/>
    <xsd:element name="num_assoc_data_elements" type="bs1:b3"/>
    <xsd:element name="num_valid_cc_elements" type="bs1:b4"/>
    <xsd:element name="mono_mixdown_present" type="bs1:b3"/>
    <xsd:element name="mono_mixdown_element_number" type="bs1:b4"/>
    <xsd:element name="stereo_mixdown_present" type="bs1:b1"/>
    <xsd:element name="stereo_mixdown_element_number" type="bs1:b4"/>
    <xsd:element name="matrix_mixdown_idx_present" type="bs1:b1"/>
    <xsd:element name="matrix_mixdown_idx" type="bs1:b2"/>
    <xsd:element name="pseudo_surround_enabled" type="bs1:b1"/>
  </xsd:sequence>
  <xsd:sequence bs2:nOccurs="sls:num_front_channel_elements" minOccurs="0" maxOccurs="unbounded" >
    <xsd:element name="front_element_is_cpe" type="bs1:b1" />
    <xsd:element name="front_element_tag_select" type="bs1:b4"/>
  </xsd:sequence>
  <xsd:sequence bs2:nOccurs="sls:num_side_channel_elements" minOccurs="0" maxOccurs="unbounded" >
    <xsd:element name="side_element_is_cpe" type="bs1:b1"/>
    <xsd:element name="side_element_tag_select" type="bs1:b4"/>
  </xsd:sequence>
  <xsd:sequence bs2:nOccurs="sls:num_back_channel_elements" minOccurs="0" maxOccurs="unbounded" >
    <xsd:element name="back_element_is_cpe" type="bs1:b1"/>
    <xsd:element name="back_element_tag_select" type="bs1:b4"/>
  </xsd:sequence>
  <xsd:element name="lfe_element_tag_select" type="bs1:b4"/>
  <xsd:element name="assoc_data_element_tag_select" type="bs1:b4"/>
</xsd:complexType>
maxOccurs="unbounded" />
<xsd:element name="cc_element_is_ind_sw" type="bs1:b1" />
<xsd:element name="valid_cc_element_tag_select" type="bs1:b4" />
</xsd:sequence>
<xsd:element ref="byte_alignment"/>
<xsd:element name="comment_field_bytes" type="bs1:b8" />
<xsd:element name="comment_field_data" type="bs1:b8" bs2:nOccurs="sls:comment_field_bytes" minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:group>

<xsd:group name="adts_sequence">
<xsd:sequence>
<xsd:element name="adts_frame" bs2:ifNext="FFF0 FFFF" maxOccurs="unbounded" >
<xsd:complexType>
<xsd:sequence>
<xsd:group ref="adtsFixedHeader"/>
<xsd:group ref="adtsVariableHeader"/>
<xsd:element name="payload">
<xsd:simpleType>
<xsd:restriction base="bs1:byteRange">
<xsd:annotation>
<xsd:appinfo>
<bs2:startCode value="FFF0 FFFF"/>
</xsd:appinfo>
<xsd:restriction>
</xsd:simpleType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:group>
</xsd:sequence>
</xsd:group>

<xsd:group name="adtsFixedHeader">
<xsd:sequence>
<xsd:element name="syncword" type="bs1:b12"/>
<xsd:element name="ID" type="bs1:b1"/>
<xsd:element name="layer" type="bs1:b2"/>
<xsd:element name="protection_absent" type="bs1:b1"/>
<xsd:element name="profileObjectType" type="bs1:b2"/>
<xsd:element name="sampling_frequency_index" type="bs1:b4"/>
<xsd:element name="private_bit" type="bs1:b17"/>
<xsd:element name="channel_configuration" type="bs1:b3"/>
<xsd:element name="original_copy" type="bs1:b1"/>
<xsd:element name="home" type="bs1:b1"/>
</xsd:sequence>
</xsd:group>

<xsd:group name="adtsVariableHeader">
<xsd:sequence>
<xsd:element name="copyright_identification_bit" type="bs1:b1"/>
<xsd:element name="copyright_identification_start" type="bs1:b1"/>
<xsd:element name="aac_frame_length" type="bs1:b13"/>
<xsd:element name="adts_buffer_fullness" type="bs1:b11"/>
<xsd:element name="number_of_raw_data_blocks_in_frame" type="bs1:b2"/>
</xsd:sequence>
</xsd:group>
<xsd:group name="raw_data_block">
<xsd:sequence>
<xsd:choice maxOccurs="unbounded" bs2:ifNext="&ID_SCE; &ID_FIL;">
<xsd:element ref="coupling_channel_element" bs2:ifNext="&ID_CCE;"/>
<xsd:element ref="channel_pair_element" bs2:ifNext="&ID_CPE;"/>
<xsd:element ref="data_stream_element" bs2:ifNext="&ID_DSE;"/>
<xsd:element ref="fill_element" bs2:ifNext="&ID_FIL;"/>
<xsd:element ref="lfe_channel_element" bs2:ifNext="&ID_LFE;"/>
<xsd:element ref="program_config_element" bs2:ifNext="&ID_PCE;"/>
<xsd:element ref="single_channel_element" bs2:ifNext="&ID_SCE;"/>
</xsd:choice>
</xsd:sequence>
</xsd:group>
<xsd:complexType name="Element">
<xsd:sequence>
<xsd:element name="id_syn_ele" type="bs1:b3"/>
<xsd:element name="element_instance_tag" type="bs1:b4"/>
<xsd:element name="replaceME" type="bs1:b1"/>
</xsd:sequence>
</xsd:complexType>

<xsd:element name="coupling_channel_element"/>
<xsd:element name="channel_pair_element"/>
<xsd:element name="data_stream_element"/>
<xsd:element name="fill_element"/>
<xsd:element name="lfe_channel_element"/>
<xsd:element name="program_config_element"/>
<xsd:element name="single_channel_element"/>

<xsd:element name="byte_alignment" type="bs1:align8"/>!
--- zzz I cant find the definition of this syntax element. -->

<xsd:simpleType name="b72">
<xsd:restriction base="xsd:nonNegativeInteger">
<xsd:maxExclusive value="4722366482869645213696"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
Listing 1.3: XSLT Stylesheet to annotate SLS BSDL data with RDO metadata

```xml
<xsl:stylesheet
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xmlns:sls="urn:ISO/IEC-14496-3:2007"
    xmlns="urn:ISO/IEC-14496-3:2007"
    xpath=default
    namespace="urn:ISO/IEC-14496-3:2007" >

    <xsl:output indent="yes" method="xml" omit-xml-declaration="yes"/>

    <xsl:param name="sampleFrequency"/>
    <xsl:variable name="frameDuration" select="1024 div $sampleFrequency"/>

    <xsl:template match="SLS" exclude-result-prefixes="#all">
      <xsl:message select="'SLS sample frequency', $sampleFrequency, ' Hz, = frame duration', $frameDuration, ' s'"/>
    </xsl:message>
    <xsl:comment>
      ...
    </xsl:comment>
    <xsl:text>&amp;#10;</xsl:text>
    <xsl:copy>
      <xsl:apply-templates select="."/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="lle:channel_pair_element">
    <xsl:variable name="id" select="count(preceding-sibling::lle:channel_pair_element)"/>
    <xsl:copy>
      <xsl:attribute name="rdo:groupID" select="$id"/>
      <xsl:attribute name="rdo:timestamp" select="$id * $frameDuration"/>
      <xsl:attribute name="rdo:duration" select="$frameDuration"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="lle:individual_channel_stream">
    <xsl:variable name="payload" select="tokenize(payload,' ')"/>
    <xsl:variable name="offset" select="number($payload[1]) - 2"/>
    <xsl:variable name="length" select="number($payload[2]) + 2"/>
    <xsl:copy>
      <xsl:attribute name="rdo:offset" select="$offset"/>
      <xsl:attribute name="rdo:length" select="$length"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="@* | *">
    <xsl:copy-of select="."/>
  </xsl:template>

</xsl:stylesheet>
```
Listing 1.4. SWRL rules to generate Semantic Distortion based on select CYC features

```swrl
/* Semantic Mapping rules */

smap:Animation

/** Semantic Mapping rules */
cyc:VisualCommunicating(?video) ∧ t:DateTimeInterval(?dti) ∧
dolce:HAPPENS−AT(?video, ?dti) ∧ cyc:imageDepicts(?video,?
cyc:Animation−MovingImage) ∧ swrlx:makeOWLThing(?sd, ?dti) ∧
swrlx:makeOWLThing(?sdq, ?dti)
→
sd:hasDistortion(?video, ?sd) ∧ sd:SemanticDistortion(?sd) ∧
do:HAS−QUALE(?sd, ?sdq) ∧ sd:Multiplicative) ∧ do:HAS−QUALE(?
sd, ?sdq) ∧ sd:ContinuousQuale(?sdq) ∧
sd:hasMagnitude(?sdq, 1)

smap:durationToOwlTime

swrlb:add(?end, ?ts, ?dur) ∧ swrlx:makeOWLThing(?e, ?x) ∧
swrlx:makeOWLThing(?Edtd, ?x)
→
t:Instant(?e) ∧ t:DateTimeDescription(?Edtd) ∧ t:hasEnd(?pi, ?e) ∧ t:inDateTime(?e,?
?Edtd) ∧ t:second(?Edtd, ?end)

smap:English

cyc:Communicating(?com) ∧ t:DateTimeInterval(?dti) ∧ dolce:HAPPENS−AT(?com, ?dti) ∧
cyc:languageUsed(?com, cyc:EnglishLanguage) ∧ swrlx:makeOWLThing(?sd, ?dti) ∧
swrlx:makeOWLThing(?sdq, ?dti)
→
sd:hasDistortion(?com, ?sd) ∧ sd:SemanticDistortion(?sd) ∧ do:HAS−QUALE(?
sd, ?sdq) ∧ sd:Multiplicative) ∧ do:HAS−QUALE(?
sd, ?sdq) ∧ sd:ContinuousQuale(?sdq) ∧
sd:hasMagnitude(?sdq, 2)

smap:Interviewing

cyc:VisualCommunicating(?video) ∧ t:DateTimeInterval(?dti) ∧
dolce:HAPPENS−AT(?video, ?dti) ∧ cyc:imageDepicts(?video, ?i) ∧
cyc:Interviewing(?i) ∧ swrlx:makeOWLThing(?sd, ?dti) ∧ swrlx:makeOWLThing(?sdq,?
?dti)
→
sd:hasDistortion(?video, ?sd) ∧ sd:SemanticDistortion(?sd) ∧ do:HAS−QUALE(?
sd, ?sdq) ∧
sd:Multiplicative) ∧ dolce:HAS−QUALE(?sd, ?sdq) ∧ sd:ContinuousQuale(?sdq) ∧ sd:hasMagnitude(?sdq, 0.75)

/***************************************************************************
+ smap:mapSDvalues
***************************************************************************/
ots:intersects(?xTime, ?yTime) ∧ dolce:HAPPENS−AT(?x, ?xTime) ∧ 
dolce:HAPPENS−AT(?y, ?yTime) ∧ sd:hasDistortion(?x, ?d)
→
sd:hasDistortion(?y, ?d)

/***************************************************************************
+ smap:NotInEnglish
***************************************************************************/
cyc:Communicating(?com) ∧ t:DateTimeInterval(?dti) ∧ dolce:HAPPENS−AT(?com, ?dti) ∧ 
cyc:languageUsed(?com, cyc:JapaneseLanguage) ∧ swrlx:makeOWLThing(?sd, ?dti) 
∧ swrlx:makeOWLThing(?sdq, ?dti)
→
sd:hasDistortion(?com, ?sd) ∧ sd:SemanticDistortion(?sd) ∧ dolce:HAS−QUALE(?sd, 
sd:Multiplicative) ∧ dolce:HAS−QUALE(?sd, ?sdq) ∧ sd:ContinuousQuale(?sdq) ∧ 
sd:hasMagnitude(?sdq, 0.5)

/***************************************************************************
+ smap:stillImage
***************************************************************************/
cyc:VisualCommunicating(?video) ∧ t:DateTimeInterval(?dti) ∧ 
swrlx:makeOWLThing(?sd, ?dti) ∧ swrlx:makeOWLThing(?sdq, ?dti)
→
sd:hasDistortion(?video, ?sd) ∧ sd:SemanticDistortion(?sd) ∧ dolce:HAS−QUALE(?sd, 
sd:Multiplicative) ∧ dolce:HAS−QUALE(?sd, ?sdq) ∧ sd:ContinuousQuale(?sdq) ∧ 
sd:hasMagnitude(?sdq, 0.5)
Semantic Distortion Ontology rules

sd:inferredOffset


⇒

sd:hasOffset(?sd:x, ?sd:xOffset)

sd:inverseImmediatePredecessor

sd:hasImmediatePredecessor(?sd:y, ?sd:x)⇒sd:hasImmediateSuccessor(?sd:x, ?sd:y)

sd:inverseImmediateSuccessor

sd:hasImmediateSuccessor(?sd:x, ?sd:y)⇒sd:hasImmediatePredecessor(?sd:y, ?sd:x)

sd:inversePredecessor

sd:hasPredecessor(?sd:y, ?sd:x)⇒sd:hasSuccessor(?sd:x, ?sd:y)

sd:inverseSuccessor

sd:hasSuccessor(?sd:x, ?sd:y)⇒sd:hasPredecessor(?sd:y, ?sd:x)
// ******************************************************************************
// * OWL Time rules
// ******************************************************************************

// ots:beforeHMS
// ******************************************************************************
t:Instant(?ots:start) \& t:Instant(?ots:end) \& t:inDateTime(?ots:start, ?ots:sDTD) \&
   t:inDateTime(?ots:end, ?ots:eDTD) \& t:hours(?ots:sDTD, ?ots:sH) \&
   swrlb:lessThan(?ots:sAsS, ?ots:eAsS)
  \rightarrow t:before(?ots:start, ?ots:end)

// ots:beforeSeconds
// ******************************************************************************
t:Instant(?ots:start) \& t:Instant(?ots:end) \& t:inDateTime(?ots:start, ?ots:sDTD) \&
   t:inDateTime(?ots:end, ?ots:eDTD) \& t:second(?ots:sDTD, ?ots:sS) \&
  \rightarrow t:before(?ots:start, ?ots:end)

// ots:contains
// ******************************************************************************
t:intervalDuring(?ots:x, ?ots:y) \rightarrow t:intervalContains(?ots:y, ?ots:x)

// ots:during
// ******************************************************************************
t:ProperInterval(?ots:X) \& t:ProperInterval(?ots:Y) \&
   t:hasBeginning(?ots:X, ?ots:Xstart) \&
   t:hasEnd(?ots:X, ?ots:Xend) \&
   t:hasBeginning(?ots:Y, ?ots:Ystart) \&
   t:hasEnd(?ots:Y, ?ots:Yend) \&
   t:before(?ots:Ystart, ?ots:Xstart) \&
   t:before(?ots:Xend, ?ots:Yend)
  \rightarrow t:intervalDuring(?ots:X, ?ots:Y)
Listings for Intelligent Media Delivery

/* *******************************************************************************/
/* ots:equivalentInstant */
/* *******************************************************************************/
t:instant(?ots:a) \& t:instant(?ots:b) \& t:inDateTime(?ots:a, ?ots:aDTD) \&
\rightarrow
sameAs(?ots:a, ?ots:b)

/* *******************************************************************************/
/* ots:equivalentInterval */
/* *******************************************************************************/
t:hasBeginning(?ots:Y, ?ots:Ystart) \& sameAs(?ots:Xstart, ?ots:Ystart) \&
t:hasEnd(?ots:X, ?ots:Xend) \& t:hasEnd(?ots:Y, ?ots:Yend) \& sameAs(?ots:Xend, ?ots:Yend)
\rightarrow
sameAs(?ots:X, ?ots:Y)

/* *******************************************************************************/
/* ots:finishedBy */
/* *******************************************************************************/

/* *******************************************************************************/
/* ots:finishes */
/* *******************************************************************************/
t:hasEnd(?ots:X, ?ots:Xend) \& t:hasEnd(?ots:Y, ?ots:Yend) \& sameAs(?ots:Xend, ?ots:Yend)
\rightarrow

/* *******************************************************************************/
/* ots:instantDuring */
/* *******************************************************************************/
\rightarrow
t:intervalDuring(?ots:X, ?ots:Y)
/*********************************************************************** 
* ots:intersects1
*********************************************************************** 
* ots:intersects1
*********************************************************************** 

/*********************************************************************** 
* ots:intersects2
*********************************************************************** 
* ots:intersects2
*********************************************************************** 

/*********************************************************************** 
* ots:intersects3
*********************************************************************** 
* ots:intersects3
*********************************************************************** 

/*********************************************************************** 
* ots:intersects4
*********************************************************************** 
* ots:intersects4
*********************************************************************** 

/*********************************************************************** 
* ots:intersects5
*********************************************************************** 
* ots:intersects5
*********************************************************************** 

/*********************************************************************** 
* ots:intersects6
*********************************************************************** 
* ots:intersects6
*********************************************************************** 

/*********************************************************************** 
* ots:intersects7
*********************************************************************** 
* ots:intersects7
*********************************************************************** 

/*********************************************************************** 
* ots:intersects8
*********************************************************************** 
* ots:intersects8
*********************************************************************** 

/*********************************************************************** 
* ots:intersects9
*********************************************************************** 
* ots:intersects9
*********************************************************************** 

/*********************************************************************** 
* ots:overlappedBy
*********************************************************************** 
* ots:overlappedBy
*********************************************************************** 
/*******************************************************************************************************************************************/
* ots:overlaps
*******************************************************************************************************************************************/
t:before(?ots:Xend, ?ots:Yend) →
t:intervalOverlaps(?ots:X, ?ots:Y)

/***************************************************************************/
* ots:startedBy
***************************************************************************/

/***************************************************************************/
* ots:starts
***************************************************************************/
t:hasBeginning(?ots:Y, ?ots:Ystart) ∧ sameAs(?ots:Xstart, ?ots:Ystart) ∧
t:intervalStarts(?ots:X, ?ots:Y)
Thank you for agreeing to take part in this experiment. It is part of research work for my PhD in Computer Engineering, with the University of Wollongong.

The experiment will take approximately 30 minutes to complete.

I want to test different methods to more effectively convey multimedia to end users under poor conditions. You will watch a number of short video clips organised into pairs. Both clips in a pair show the same footage, but use a different method for improving the content. All of the clips will be distorted to some degree, so they’re not what you’d like to see, but this is important for the test.

Please indicate which clip (A or B) best conveys the gist of the news article to you. (There is no right answer, I am interested in your perception of the clips.)

Your decision must reflect your opinion of the overall combined audio and video content. Observe and listen carefully to the entire clip before making your choice.

During the test you will see the same articles multiple times to test different conditions. You do not have to remember anything from one pair to the next. Just indicate which clip in the current pair seems better to you.

If desired, you can watch a clip more than once, or repeat clip A after watching clip B (et cetera). Once you have decided which clip best conveys the gist of the article, tell me your decision (A or B), and move on to the next pair.

If you have any trouble with the controls, I can assist you.
Appendix J

Listings for Reconfigurable Media Coding

Listing J.1: BS Schema describing MPEG-4 Visual Simple Profile

<!−−*************************************************************************→
<!−− Digital Item Adaptation ISO/IEC 21000−7 −−>
<!−− Schema for the MPEG−4 Visual syntax −−>
<!−− Created by Ghent University − Multimedia Lab −−>
<!−− Modified by University of Wollongong −−>
<!−−*************************************************************************→
<!DOCTYPE bsschema SYSTEM "entities.dtd">
<xsd:schema xmlns:m4v="MPEG−4_Visual" xmlns="MPEG−4_Visual"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bs2xs="urn:mpeg:mpeg21:2003:01−DIA−BSDL2x−NS"
xmlns:bs1xs="urn:mpeg:mpeg21:2003:01−DIA−BSDL1x−NS"
xmlns:bs2s="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS"
xmlns:bs0s="urn:mpeg:mpeg21:2003:01−DIA−BSDL0−NS"
targetNamespace="MPEG−4_Visual" elementFormDefault="qualified"
bs2:rootElement="m4v:bitstream" bs2:defaultTreeInMemory="false"
xmlns:rmc="urn:mpeg:2006:01−RMC−NS">
<xsd:annotation>
<xsd:appinfo>
<bs1xs:script ref="functions.js"/>
<rml:demultiplexer name="MBtoBlocks" inputName="MB">
<rml:output name="Y"/>
<rml:output name="U"/>
<rml:output name="V"/>
</rml:demultiplexer>
</xsd:appinfo>
</xsd:annotation>
<xsd:import namespace="urn:mpeg:mpeg21:2003:01−DIA−BSDL1−NS"
schemaLocation="/BSDL/schemata/BSDL1.xsd"/>
<xsd:include schemaLocation="mpeg−4_visual_simple_types.xsd"/>
<!−−*************************************************************************→
<xsd:element name="bitstream">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="bitstreamObject" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:element ref="visual_object_sequence"/>
                        <xsd:element name="GOVOP" type="GroupOfVideoObjectPlaneType"/>
                        <xsd:element name="VOP" type="VideoObjectPlaneType"/>
                        <xsd:element ref="unsupportedBitstream"/>
                    </xsd:choice>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute ref="bs1:bitstreamURI"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="visual_object_sequence">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="visual_object_sequence_start_code" type="m4v:StartCodeType"/>
            <xsd:element name="profile_and_level_indication" type="m4v:hex1"/>
            <xsd:element ref="m4v:user_data" minOccurs="0"/>
            <xsd:element name="visual_object" type="m4v:VisualObjectType"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="visual_object">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="visual_object_start_code" type="m4v:StartCodeType"/>
            <xsd:element name="is_visual_object_identifier" type="bs1:b1" bs0:variable="true">bs2:x:variable name="m4v:visual_object_verid" value="1"/></xsd:element>
        </xsd:sequence>
        <xsd:annotation>
            <xsd:appinfo>
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:complexType>
</xsd:element>
<xsd:element name="user_data" bs2:ifNext="&udtaSC;">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="user_data_start_code" type="m4v:StartCodeType"/>
      <xsd:element name="user_data_payload" type="m4v:NextStartCodeType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- 6.2.3 Video Object Layer  -->
<!-- 6.2.3 Video Object Layer  -->
<xsd:complexType name="VideoObjectLayerType">
  <xsd:choice>
    <xsd:group ref="long_video_header" bs2:ifNext="&volSC;"/>
    <xsd:group ref="short_video_header"/>
  </xsd:choice>
</xsd:complexType>
<xsd:group name="long_video_header">
  <xsd:element name="video_object_layer_start_code" type="m4v:StartCodeType">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2x:variable name="shortVideoHeader" value="0"/>
        <bs2x:variable name="volVersion" value="if ($m4v:visual_object_verid) then $m4v:visual_object_verid else 1"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="random accessible_vol" type="bs1:b1"/>
  <xsd:element name="video_object_type_indication" type="bs1:b8"/>
  <!-- Ignoring FGS fields -->
  <xsd:element name="is_object_layer_identifier" type="bs1:b1" bs0:variable="true"/>
  <xsd:sequence minOccurs="0" bs2:if="$m4v:is_object_layer_identifier = 1">"/>
  <xsd:element name="video_object_layer_verid" type="bs1:b4">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2x:variable name="volVersion" value="/text()"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="video_object_layer_priority" type="bs1:b3"/>
</xsd:sequence>
  <xsd:element name="aspect_ratio_info" type="bs1:b4" bs0:variable="true"/>
  <xsd:sequence minOccurs="0" bs2:if="&extendedPAR;">
    <xsd:element name="par_width" type="bs1:b8"/>
    <xsd:element name="par_height" type="bs1:b8"/>
  </xsd:sequence>
  <xsd:element name="vol_control_parameters" type="bs1:b1" bs0:variable="true"/>
  <xsd:sequence minOccurs="0" bs2:if="$m4v:vol_control_parameters = 1">"/>
  <xsd:element name="chroma_format" type="bs1:b2"/>
  <xsd:element name="low_delay" type="bs1:b1"/>
  <xsd:element name="vbv_parameters" type="bs1:b1" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vbv_parameters = 1">
  <xsd:element name="first_half_bit_rate" type="bs1:b15"/>
  <xsd:element name="marker_bit00" type="bs1:b1" fixed="1"/>
  <xsd:element name="latter_half_bit_rate" type="bs1:b15"/>
  <xsd:element name="marker_bit01" type="bs1:b1" fixed="1"/>
  <xsd:element name="first_half_vbv_buffer_size" type="bs1:b15"/>
  <xsd:element name="marker_bit02" type="bs1:b1" fixed="1"/>
  <xsd:element name="latter_half_vbv_buffer_size" type="bs1:b3"/>
  <xsd:element name="first_half_vbv_occupancy" type="bs1:b11"/>
  <xsd:element name="marker_bit03" type="bs1:b1" fixed="1"/>
  <xsd:element name="latter_half_vbv_occupancy" type="bs1:b15"/>
  <xsd:element name="marker_bit04" type="bs1:b1" fixed="1"/>
</xsd:sequence>

<xsd:element name="video_object_layer_shape" type="bs1:b2" bs0:variable="true">
  <xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape = &grayscale; and $volVersion != 1">
    <xsd:element name="video_object_layer_shape_extension" type="bs1:b4"/>
  </xsd:sequence>
  <xsd:annotation>
    <xsd:appinfo>
      <bs2x:variable name="vopTimeIncrementBits" value="bs1x:numBits(./text())"/>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>

<xsd:element name="marker_bit05" type="bs1:b1" fixed="1"/>
<xsd:element name="vop_time_increment_resolution" type="bs1:b16"/>
<xsd:annotation>
  <xsd:appinfo>
    <bs2x:variable name="vopTimeIncrementBits" value="bs1x:numBits(./text())"/>
  </xsd:appinfo>
</xsd:annotation>
</xsd:element>

<xsd:element name="marker_bit06" type="bs1:b1" fixed="1"/>
<xsd:element name="fixed_vop_rate" type="bs1:b1"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:fixed_vop_rate = 1">
  <xsd:element name="fixed_vop_time_increment" type="m4v:VOPTimeIncrementType"/>
</xsd:sequence>
<xsd:choice>
  <xsd:element name="nonBinaryShapeData" type="nonBinaryShapeDataType" bs2:if="$m4v:video_object_layer_shape != &binaryOnly;"/>
  <xsd:element name="BinaryShapeData" type="BinaryShapeDataType"/>
</xsd:choice>
<xsd:element name="next_start_code" type="bs1:align8" minOccurs="0"/>
<xsd:element name="user_data" minOccurs="0"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:sprite_enable = &static;">
  <xsd:element name="unsupportedBitstream01" type="xsd:base64Binary"/>
</xsd:sequence>
<xsd:element name="unsupportedBitstream01" type="xsd:base64Binary"/>
</xsd:complexType>

<xsd:complexType name="nonBinaryShapeDataType">
  <xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape = &rectangular;">
    <xsd:element name="marker_bit07" type="bs1:b1" fixed="1"/>
    <xsd:element name="video_object_layer_width" type="bs1:b13"/>
    <xsd:element name="video_object_layer_width" type="bs1:b13"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="marker_bit08" type="bs1:b1" fixed="1"/>
<xsd:element name="video_object_layer_height" type="bs1:b13" bs0:variable="true"/>
<xsd:element name="marker_bit09" type="bs1:b1" fixed="1"/>
<xsd:annotation>
  <xsd:appinfo>
    <bs2x:variable name="mbCount"
      value="(($m4v:video_object_layer_width + 15) idiv 16) *
        (($m4v:video_object_layer_height + 15)
          idiv 16)="/n
      <bs2x:variable name="mbNumberLength"
        value="bs1x:numBits($mbCount)="/n
  </xsd:appinfo>
</xsd:annotation></xsd:element>
<xsd:element name="interlaced" type="bs1:b1" bs0:variable="true"/>
<xsd:element name="obmc_disable" type="bs1:b1"/>
<xsd:element name="sprite_enable" type="SpriteType" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:sprite_enable = &static; or
  $m4v:sprite_enable = &GMC;="/n
  <xsd:element name="unsupportedBitstream02" type="xsd:base64Binary="/n
  <!-- ignoring sprite fields -->
</xsd:sequence>
<xsd:element name="sadct_disable" type="bs1:b1"/>
<xsd:element name="not_8_bit" type="bs1:b1" bs0:variable="true"/>
<xsd:appinfo>
  <bs2x:variable name="m4v:quant_precision" value="5="/n
</xsd:appinfo>
<xsd:annotation>
</xsd:element>
<xsd:element name="quant_precision" type="bs1:b4" bs0:variable="true"/>
<xsd:element name="bits_per_pixel" type="bs1:b4="/n
</xsd:element>
<xsd:sequence minOccurs="0" bs2:if="$m4v:not_8_bit = 1="/n
  <xsd:element name="no_gray_quant_update" type="bs1:b1="/n
</xsd:element>
<xsd:element name="composition_method" type="bs1:b1="/n
</xsd:element>
<xsd:element name="linear_composition" type="bs1:b1="/n
</xsd:element>
<xsd:sequence minOccurs="0" bs2:if="$m4v:quant_type = 1="/n
  <xsd:element name="load_intra_quant_mat" type="bs1:b1" bs0:variable="true="/n
</xsd:element>
<xsd:sequence minOccurs="0" bs2:if="$m4v:load_intra_quant_mat = 1="/n
  <xsd:element name="intra_quant_mat" type="QuantisationMatrixType="/n
</xsd:element>
<xsd:sequence>
  <xsd:element name="load_nonintra_quant_mat" type="bs1:b1"
    bs0:variable="true="/n
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:load_nonintra_quant_mat = 1="/n

<xsd:element name="nonintra_quant_mat" type="QuantisationMatrixType"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape = &grayscale;">
<xsd:element name="unsupportedBitstream03" type="xsd:base64Binary"/>
<!−− ignoring grayscale quantisation matrices −−>
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$volVersion != 1">
<xsd:element name="unsupportedBitstream04" type="xsd:base64Binary"/>
<!−− ignore scalability −−>
</xsd:sequence>
</xsd:sequence>
<xsd:element name="complexity_estimation_disable" type="bs1:b1" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:complexity_estimation_disable = 0">
<xsd:element name="define_vop_complexity_estimation_header" type="DefineVOPComplexityEstimationHeaderType"/>
<xsd:annotation>
<xsd:appinfo>
<bs2x:variable name="defineVopComplexityEstimationHeader"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:sequence>
</xsd:element>
</xsd:sequence>
<xsd:element name="resync_marker_disable" type="bs1:b1"/>
<xsd:element name="data_partitioned" type="bs1:b1" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:data_partitioned = 1">
<xsd:element name="reversible_vlc" type="bs1:b1"/>
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$volVersion != 1">
<xsd:element name="newpred_enable" type="bs1:b1" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:newpred_enable = 1">
<xsd:element name="requested_upstream_message_type" type="bs1:b2"/>
</xsd:sequence>
</xsd:sequence>
<xsd:element name="newpred_segment_type" type="bs1:b1"/>
</xsd:sequence>
<xsd:element name="reduced_resolution_vop_enable" type="bs1:b1" bs0:variable="true"/>
</xsd:sequence>
<xsd:element name="scalability" bs1:b1" bs0:variable="true"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:scalability = 1">
<xsd:element name="unsupportedBitstream04" type="xsd:base64Binary"/>
<!−− ignore scalability −−>
</xsd:sequence>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="BinaryShapeDataType">
<xsd:sequence minOccurs="0" bs2:if="$volVersion != 1">
<xsd:element name="scalability" type="bs1:b1"/>
</xsd:sequence>
<xsd:annotation>
<xsd:appinfo>
<bs2x:variable name="scalability" value="/text()"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:complexType>
<xsd:element name="unsupportedBitstream05" type="xsd:base64Binary"/>

<!-- ignore scalability -->

<xsd:element name="resync_marker_disable" type="bs1:b1"/>

<xsd:complexType name="DefineVOPComplexityEstimationHeaderType">

<xsd:element name="estimation_method" type="bs1:b2" bs0:variable="true"/>

<xsd:sequence minOccurs="0" bs2:if="$m4v:estimation_method = 0 or $m4v:shape_complexity_estimation_disable = 0" bs0:variable="true"/>

<xsd:element name="opaque" type="bs1:b1"/>

<xsd:element name="transparent" type="bs1:b1"/>

<xsd:element name="intra_cae" type="bs1:b1"/>

<xsd:element name="inter_cae" type="bs1:b1"/>

<xsd:element name="no_update" type="bs1:b1"/>

<xsd:element name="upsampling" type="bs1:b1"/>

<xsd:sequence minOccurs="0" bs2:if="$m4v:texture_complexity_estimation_set_1_disable = 0" bs0:variable="true"/>

<xsd:element name="intra_blocks" type="bs1:b1"/>

<xsd:element name="inter_blocks" type="bs1:b1"/>

<xsd:element name="inter4v_blocks" type="bs1:b1"/>

<xsd:element name="not_coded_blocks" type="bs1:b1"/>

<xsd:sequence irresistibles="0"/>

<xsd:element name="dct_coefs" type="bs1:b1"/>

<xsd:element name="dct_lines" type="bs1:b1"/>

<xsd:element name="vlc_symbols" type="bs1:b1"/>

<xsd:element name="vlc_bits" type="bs1:b1"/>

<xsd:sequence minOccurs="0" bs2:if="$m4v:motion_compensation_complexity_disable = 0" bs0:variable="true"/>

<xsd:element name="apm" type="bs1:b1"/>

<xsd:element name="npm" type="bs1:b1"/>

<xsd:element name="interpolate_mc_q" type="bs1:b1"/>

<xsd:element name="forw_back_mc_q" type="bs1:b1"/>

<xsd:element name="halfpel2" type="bs1:b1"/>
<xsd:element name="halfpel4" type="bs1:b1"/>
</xsd:sequence>
<xsd:element name="marker_bit11" type="bs1:b1"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:estimation_method = 1">
  <xsd:element name="version2_complexity_estimation_disable" type="bs1:b1">
    bs0:variable="true"/>
  <xsd:sequence minOccurs="0" bs2:if="$m4v:version2_complexity_estimation_disable = 1">
    <xsd:element name="sadct" type="bs1:b1"/>
    <xsd:element name="quarterpel" type="bs1:b1"/>
  </xsd:sequence>
</xsd:sequence>
</xsd:sequence>
<xsd:sequence>
</xsd:sequence>
<xsd:sequence>
</xsd:sequence>
</xsd:complexType>

<!-- <xsd:complexType name="ReadVOPComplexityEstimationHeaderType" >
<xsd:sequence>
  <xsd:element name="estimation_method" type="bs1:b2" />
  <xsd:sequence minOccurs="0" bs2:if="m4v:estimation_method = 0 or m4v:estimation_method = 1">
    <xsd:element name="shape_complexity_estimation_disable" type="bs1:b1"/>
  </xsd:sequence>
  <xsd:sequence minOccurs="0" bs2:if="m4v:shape_complexity_estimation_disable = 0">
    <xsd:element name="opaque" type="bs1:b1"/>
    <xsd:element name="transparent" type="bs1:b1"/>
    <xsd:element name="intra_cae" type="bs1:b1"/>
    <xsd:element name="inter_cae" type="bs1:b1"/>
    <xsd:element name="no_update" type="bs1:b1"/>
    <xsd:element name="upsampling" type="bs1:b1"/>
  </xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="m4v:texture_complexity_estimation_set_1_disable = 0">
  <xsd:element name="intra_blocks" type="bs1:b1"/>
  <xsd:element name="inter_blocks" type="bs1:b1"/>
  <xsd:element name="inter4v_blocks" type="bs1:b1"/>
  <xsd:element name="not_coded_blocks" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="m4v:texture_complexity_estimation_set_2_disable = 0">
  <xsd:element name="dct_coefs" type="bs1:b1"/>
  <xsd:element name="dct_lines" type="bs1:b1"/>
  <xsd:element name="vlc_symbols" type="bs1:b1"/>
  <xsd:element name="vlc_bits" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="m4v:motion Compensation_complexity_disable = 0">
  <xsd:element name="apm" type="bs1:b1"/>
  <xsd:element name="npm" type="bs1:b1"/>
  <xsd:element name="interpolate_mc_q" type="bs1:b1"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="forw_back_mc_q" type="bs1:b1"/>
<xsd:element name="halfpel2" type="bs1:b1"/>
<xsd:element name="halfpel4" type="bs1:b1"/>
</xsd:sequence>
<xsd:element name="marker_bit" type="bs1:b1"/>
<xsd:sequence minOccurs="0" bs2:if="m4v:estimation_method = 1">
<xsd:element name="version2_complexity_estimation_disable"/>
<xsd:sequence minOccurs="0" bs2:if="m4v:version2_complexity_estimation_disable = 1">
<xsd:element name="sadct" type="bs1:b1"/>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="QuantisationMatrixType">
<xsd:sequence>
<xsd:element name="quant_mat" type="xsd:unsignedByte" maxOccurs="63" bs2:ifNext="01 FF"/>
<xsd:element name="final_quant_mat" type="xsd:unsignedByte"/>
</xsd:sequence>
</xsd:complexType>

<!−− 6.2.4 Group of Video Object Plane −−>
<xsd:complexType name="GroupOfVideoObjectPlaneType">
<xsd:sequence>
<xsd:element name="group_of_vop_start_code" type="m4v:StartCodeType" fixed="&goVopSC;"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="time_code" type="bs1:b18"/>
<xsd:element name="closed_gov" type="bs1:b1"/>
<xsd:element name="broken_link" type="bs1:b1"/>
<xsd:element name="next_start_code" type="bs1:align8" minOccurs="0"/>
<xsd:element ref="user_data" minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>

<!-- 6.2.5 Video Object Plane -->
<!-- 6.2.5 Video Object Plane -->
<xsd:complexType name="VideoObjectPlaneType">
<xsd:sequence>
<xsd:element name="vopHeader" type="m4v:VideoObjectPlaneHeaderType"
    rmc:port="1"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coded=1">
<xsd:element name="motion_shape_texture" type="MotionShapeTextureType"/>
</xsd:sequence>
</xsd:complexType>

<!-- 6.2.5 Video Object Plane -->
<xsd:complexType name="VideoObjectPlaneHeaderType">
<xsd:sequence>
<xsd:element name="vop_start_code" type="m4v:StartCodeType" fixed="&vopSC;"/>
<xsd:element name="vop_coding_type" type="bs1:b2" bs0:variable="true"/>
<xsd:element name="modulo_time_base1" type="bs1:b1" minOccurs="0" maxOccurs="unbounded" bs2:ifNext="80 FF"/>
<xsd:element name="modulo_time_base0" type="bs1:b1"/>
<xsd:element name="marker_bit12" type="bs1:b1" fixed="1"/>
<xsd:element name="vop_time_increment" type="m4v:VOPTimeIncrementType"/>
<xsd:element name="marker_bit13" type="bs1:b1" fixed="1"/>
<xsd:element name="vop_coded" type="bs1:b1" bs0:variable="true"/>
</xsd:choice>
<xsd:element name="next_start_code" type="bs1:align8" bs2:if="$m4v:vop_coded"
    = "0"/>
</xsd:group ref="VOPData"/>
</xsd:choice>
</xsd:sequence>
</xsd:complexType>

<xsd:group name="VOPData">
<xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:newpred_enable = 1"/>
<xsd:element name="vop_id" type="VOPidType"/>
<xsd:element name="vop_id_for_prediction_indication" type="bs1:b1"
  bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_id_for_prediction_indication = 1">
  <xsd:element name="vop_id_for_prediction" type="VOPidType"/>
</xsd:sequence>
<xsd:element name="marker_bit14" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape != &binaryOnly; and
  ($m4v:vop_coding_type = &pVOP; or ($m4v:vop_coding_type = &sVOP; and
  $m4v:sprite_enable = &GMC;))">
<xsd:element name="vop_rounding_type" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape != &rectangular; and
  ($m4v:vop_coding_type = &pVOP; or $m4v:vop_coding_type = &iVOP;)">
<xsd:element name="vop_reduced_resolution" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$reduced_resolution_vop_enable = 1 and $m4v:video_object_layer_shape = &rectangular; and
  ($m4v:vop_coding_type = &pVOP; or $m4v:vop_coding_type = &iVOP;)">
</xsd:sequence>
<xsd:choice>
  <xsd:group ref="NonBinaryVOPData" bs2:if="$m4v:video_object_layer_shape != &binaryOnly;">
    <!-- Ignoring non rectangular bitstreams for now -->
  </xsd:group>
  <xsd:group name="NonBinaryVOPData"/>
</xsd:choice>
<xsd:sequence minOccurs="0" bs2:if="$m4v:complexity_estimation_disable = 0">
  <!-- Ignoring complexity estimation bitstreams for now -->
</xsd:sequence>
<xsd:element name="intra_dc_vlc_thr" type="bs1:b3" bs0:variable="true"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:interlaced = 1">
  <xsd:element name="top_field_first" type="bs1:b1"/>
</xsd:sequence>
<xsd:element name="alternate_vertical_scan_flag" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type = &sVOP; and ($m4v:sprite_enable = &static; or
  $m4v:sprite_enable = &GMC;)">
  <!-- Ignoring sprite bitstreams for now -->
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape != &binaryOnly;">
  <xsd:element name="vop_quant" type="VOPQuantType"/>
</xsd:sequence>
<xsd:annotation xmlns="xsd:appinfo">
<bs2x:variable name="resyncMarkerLength" value="17"/>
</xsd:appinfo></xsd:annotation>
</xsd:element>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape = &grayscale;">
<!— ignoring grayscale bitstreams for now ——>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type != &iVOP;">
<xsd:element name="vop_fcode_forward" type="bs1:b3" bs0:variable="true">
<xsd:annotation>
<xsd:appinfo>
<bs2x:variable name="resyncMarkerLength" value="./text() + 16"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type = &bVOP;">
<xsd:element name="vop_fcode_backward" type="bs1:b3">
<xsd:annotation>
<xsd:appinfo>
<bs2x:variable name="resyncMarkerLength" value="max(($resyncMarkerLength, 18))"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:choice minOccurs="0">
<xsd:group ref="NonScalableVOPData" bs2:if="$m4v:scalability = 0">
<!— ignoring scalable bitstreams for now ——>
</xsd:group>
</xsd:choice>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="VPType">
<xsd:sequence>
<xsd:element name="video_packet_header" type="VideoPacketHeaderType"/>
<xsd:element name="combined_motion_shape_texture" type="CombinedMotionShapeTextureType"/>
</xsd:sequence>
</xsd:complexType>
<xsd:group name="NonScalableVOPData">
<xsd:sequence>
<xsd:sequence minOccurs="0">
bs2:if="$m4v:video_object_layer_shape != &rectangular; and $m4v:vop_coding_type != &iVOP;">
<xsd:element name="vop_shape_coding_type" type="bs1:b1"/>
</xsd:sequence>
<!— ISO/IEC 14496 – 2 has VP & MotionShape texture in here, but I want to separate them from the VOP header. ——>
</xsd:sequence>
</xsd:group>
</xsd:complexType>
<!—— 6.2.5.2 Video Packet Header ——>
<!——  ************************************** ——>
<xsd:complexType name="VideoPacketHeaderType">
  <xsd:sequence>
    <xsd:element name="resync_marker" type="ResyncMarkerType"/>
  </xsd:sequence>
  <xsd:sequence minOccurs="0" bs2;if="$m4v:video_object_layer_shape != &rectangular;">
    <!— — ignoring nonrectangular bitstreams ——>
  </xsd:sequence>
  <xsd:element name="macroblock_number" type="MBNumberType"/>
  <xsd:sequence minOccurs="0" bs2;if="$m4v:video_object_layer_shape != &binaryOnly;">
    <xsd:element name="quant_scale" type="VOPQuantType"/>
  </xsd:sequence>
  <xsd:sequence minOccurs="0" bs2;if="$m4v:video_object_layer_shape = &rectangular;">
    <xsd:element name="header_extension_code" type="bs1:b1" bs0:variable="true"/>
  </xsd:sequence>
  <xsd:element name="VPHeaderExtension" minOccurs="0" bs2;if="$m4v:header_extension_code = 1" type="VPHeaderExtensionType"/>
</xsd:complexType>

<xsd:complexType name="VPHeaderExtensionType">
  <xsd:sequence>
    <xsd:element name="modulo_time_base1" type="bs1:b1" minOccurs="0" maxOccurs="unbounded" bs2;ifNext="80 FF"/>
    <xsd:element name="modulo_time_base0" type="bs1:b1"/>
    <xsd:element name="marker_bit15" type="bs1:b1"/>
    <xsd:element name="vop_time_increment" type="m4v:VOPTimeIncrementType"/>
    <xsd:element name="marker_bit16" type="bs1:b1"/>
    <xsd:element name="vop_coding_type" type="bs1:b2"/>
    <xsd:sequence minOccurs="0" bs2;if="$m4v:video_object_layer_shape != &rectangular;">
      <!— — ignoring non rectangular bitstreams for now ——>
    </xsd:sequence>
    <xsd:sequence minOccurs="0" bs2;if="$m4v:video_object_layer_shape != &binaryOnly;">
      <xsd:element name="intra_dc_vlc_thr" type="bs1:b3"/>
    </xsd:sequence>
    <xsd:sequence minOccurs="0" bs2;if="$m4v:vop_coding_type != &sVOP; and $m4v:sprite_enable = &GMC;">
      <!— — ignoring sprite bitstreams for now ——>
    </xsd:sequence>
    <xsd:sequence minOccurs="0" bs2;if="$reduced_resolution_vop_enable = 1 and $m4v:video_object_layer_shape = &rectangular; and ($m4v:vop_coding_type = &pVOP; or $m4v:vop_coding_type = &iVOP;)">
      <xsd:element name="vop_reduced_resolution" type="bs1:b1"/>
    </xsd:sequence>
    <xsd:sequence minOccurs="0" bs2;if="$m4v:vop_coding_type != &iVOP;"/>
<xsd:element name="vop_fcode_forward" type="bs1:b3"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" if="$m4v:vop_coding_type = &bVOP;">
<xsd:element name="vop_fcode_backward" type="bs1:b3"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" if="$m4v:newpred_enable = 1">
<xsd:element name="vop_id" type="VOPIdType"/>
<xsd:element name="vop_id_for_prediction_indication" type="bs1:b1" variable="true"/>
<xsd:sequence minOccurs="0" if="$m4v:vop_id_for_prediction_indication = 1">
<xsd:element name="vop_id_for_prediction" type="VOPIdType"/>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
</xsd:complexType>

<!-- 6.2.5.3 Motion Shape Texture -->
<xsd:complexType name="MotionShapeTextureType">
<xsd:choice>
<xsd:element name="unsupportedBitstream07" type="xsd:base64Binary" if="$m4v:data_partitioned = 1"/>
<xsd:element name="combined_motion_shape_texture" type="CombinedMotionShapeTextureType"/>
</xsd:choice>
</xsd:complexType>

<xsd:complexType name="CombinedMotionShapeTextureType">
<xsd:sequence>
<xsd:element ref="MB" minOccurs="1" maxOccurs="unbounded" nOccurs="$mbCount">
<ifNext="000002 FFFFFF" or validstuffingbits = 0>
<!-- <xsd:annotation>
<xsd:appinfo>
<bs0:readAhead unit="bit" offset="0" type="xsd:integer"/>
</xsd:appinfo>
</xsd:annotation> -->
</xsd:element>
</xsd:sequence>
</xsd:complexType>

<!-- 6.2.6 Macroblock -->
<xsd:element name="MB" rmc:port="2" rmc:demultiplexerInput="MBtoBlocks#MB">
<bs2x:variable name="blockNo" value="0"/>
<bs2x:variable name="m4v:not_coded" value="0"/>
<bs2x:variable name="use_intra_dc_vlc" value="1"/>
<!-- TODO look at intra_dc_thr to determine this flag -->
</xsd:appinfo>
<xsd:complexType>
<xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type != &bVOP;的家庭">
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape != &rectangular;的家庭">
<!-- ignoring non-rectangular bitstreams -->
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape != &rectangular;家庭">
<xsd:sequence minOccurs="0" bs2:if="true()">
<!-- ignoring for now -->
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="!transparent_MB()家庭">
<xsd:choice>
<xsd:group ref="NonRectangularMBHeader">
<bs2:if="$m4v:video_object_layer_shape != &rectangular;爸爸">
</xsd:choice>
</xsd:sequence>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="RectangularMB">
<xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type != &iVOP;家庭">
<xsd:sequence minOccurs="0" bs2:if="!sprite.enable = "static" && sprite.transmit_mode == "piece"爸爸">
<xsd:element name="not_coded" type="bs1:b1" bs0:variable="true"/>
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:not_coded = 0 or $m4v:vop_coding_type = &iVOP;家庭">
<!- - || (vop_coding_type == "S" && low_latency(sprite_enable && sprite.transmit_mode == "piece"))爸爸 -->
</xsd:sequence>
<xsd:element name="mcmbc" type="McbpcType">
<!−− || (vop_coding_type == "S" && low_latency_sprite_enable &&
sprite_transmit_mode == "piece") −−>
<xsd:sequence minOccurs="0" bs2:if="$m4v:vop_coding_type = &sVOP;"/>
<!−− & sprite_enable == GMC & (derived_mb_type == 0 || derived_mb_type
== 1) −−>
<!−− ignoring sprites −−>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$shortVideoHeader = 0 and ($derived_mb_type = '3' or $derived_mb_type
= '4')"/>
<xsd:element name="ac_pred_flag" type="bs1:b1"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$derived_mb_type != 'Stuffing'"/>
<xsd:element name="cbpy" type="CbpyType">
<xsd:annotation>
<xsd:appinfo>
<bs2x:variable name="pattern_code"
value="(substring(text(),1,1),substring(text(),2,1),substring(text(),3,1),
substring(text(),4,1),$pattern_code)"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:element>
<!−− swapping around return statement − rest of macroblock in here −−>
<xsd:sequence minOccurs="0" bs2:if="$derived_mb_type = '1' or
$derived_mb_type = '4'"/>
<xsd:element name="dquant" type="bs1:b2"/>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$m4v:interlaced=1"
and $m4v:field_prediction=1"−−>
<!−− FIXME ignoring interlaced for now −−>
</xsd:sequence>
<xsd:sequence minOccurs="0"
bs2:if="not($m4v:ref_select_code=3 and $m4v:scalability=1) and
$m4v:sprite_enable != &static;"/>
<xsd:sequence minOccurs="0" bs2:if="($derived_mb_type = '0' or $derived_mb_type = '1') and
($m4v:vop_coding_type = &pVOP;)"/>
<!−− or ($m4v:vop_coding_type = &sVOP; and $m4v:mcsel = 0) −−>
<xsd:element name="motion_vector_forward"
type="MotionVectorForwardType"/>
<xsd:sequence minOccurs="0" bs2:if="$m4v:interlaced=1"
and $m4v:field_prediction=1"−−>
<!−− ignoring interlaced for now −−>
</xsd:sequence>
</xsd:sequence>
<xsd:sequence minOccurs="0" bs2:if="$derived_mb_type = "2""
<bs2:if="!transparent_block()" ignoring for now −−>
<xsd:element name="motion_vector_forward2"
type="MotionVectorForwardType" minOccurs="4" maxOccurs="4"/>
</xsd:sequence>
</xsd:sequence>
</xsd:sequence>
<xsd:group ref="Blocks" minOccurs="0"/>
<xsd:complexType name="BlockType">

</xsd:complexType>
<xsd:sequence>
  <xsd:sequence minOccurs="0" bs2:if="$m4v:partitioned=0 and ($derived_mb_type = '3' or $derived_mb_type = '4')">
    <xsd:choice minOccurs="0">
      <xsd:element name="intra_dc_coefficient" type="xsd:unsignedByte" bs2:if="$shortVideoHeader=1"/>
      <xsd:group ref="BlockData" bs2:if="$use_intra_vlc = 1"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:sequence>

<xsd:sequence minOccurs="0" bs2:if="$pattern_code[$blockNo] = '1'">
  <xsd:element ref="DCTCoefficient" minOccurs="0" maxOccurs="unbounded" bs2:if="$m4v:last = '0'"/>
</xsd:sequence>

<xsd:complexType name="BlockData">
  <xsd:sequence>
    <xsd:element name="dct_dc_size" type="DCTdcSizeType" bs0:variable="true"/>
    <xsd:sequence minOccurs="0" bs2:if="$m4v:dct_dc_size != 0">
      <xsd:element name="dct_dc_differential" type="DCTdcDifferentialType"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="DCTCoefficient">
  <xsd:annotation>
    <xsd:appinfo>
      <bs2x:variable name="dctVLC" value="true()"/>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>

<xsd:sequence>
  <!-- TODO: see if i can streamline this any -->
  <xsd:element name="escape1" type="bs1:b8" bs2:ifNext="&dctESCAPE1;"/>
  <xsd:element name="escape2" type="bs1:b9" bs2:ifNext="&dctESCAPE2;"/>
  <xsd:group ref="EscapeType3" bs2:ifNext="&dctESCAPE3;"/>
</xsd:choice>
</xsd:sequence>

<xsd:sequence minOccurs="0" bs2:if="&dctVLC" minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>

<xsd:element name="DCTCoefficientGroup">
  <xsd:sequence>
    <xsd:element name="value" type="DCTCoefficientType"/>
  </xsd:annotation>
  <xsd:appinfo>
</xsd:element>
<bs2x:variable name="m4v:last" value="substring-before(/text(),',')"/>
</xsd:appinfo>
</xsd:annotation>
</xsd:element>
<xsd:element name="levelSign" type="bs1:b1"/>
</xsd:sequence>
</xsd:group>

<!-- ****************************  -->

<!-- DCT Coefficient EscapeTypeType (7.4.1.3) -->
<!-- ****************************  -->

<xsd:complexType name="EscapeType3">
  <xsd:sequence>
    <xsd:element name="escape3" type="bs1:b9" fixed="15">
      <xsd:annotation>
        <xsd:appinfo>
          <bs2x:variable name="dctVLC" value="false()"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="last" type="bs1:b1" bs0:variable="true"/>
    <xsd:element name="run" type="bs1:b6"/>
    <xsd:element name="marker_bit20" type="bs1:b1"/>
    <xsd:element name="level" type="bs1:b12"/>
    <xsd:element name="marker_bit21" type="bs1:b1"/>
  </xsd:sequence>
</xsd:complexType>

<!-- ****************************  -->

<!-- 6.2.8 Still Texture Object -->
<!-- ****************************  -->

<xsd:complexType name="StillTextureObjectType">
  <xsd:complexContent>
    <xsd:extension base="UnparsedObjectType"/>
  </xsd:complexContent>
</xsd:complexType>

<!-- ****************************  -->

<!-- 6.2.9 Mesh Object -->
<!-- ****************************  -->

<xsd:complexType name="MeshObjectType">
  <xsd:complexContent>
    <xsd:extension base="UnparsedObjectType"/>
  </xsd:complexContent>
</xsd:complexType>

<!-- ****************************  -->

<!-- 6.2.10 FBA Object -->
<!-- ****************************  -->

<xsd:complexType name="FBAObjectType">
  <xsd:complexContent>
    <xsd:extension base="UnparsedObjectType"/>
  </xsd:complexContent>
</xsd:complexType>
<! -- 6.2.11 3D Mesh Object -->
<! -- ************************************************************************ -->
<xsd:complexType name="Mesh_3D_ObjectType">
    <xsd:complexContent>
        <xsd:extension base="UnparsedObjectType"/>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="UnparsedObjectType">
    <xsd:sequence>
        <xsd:element name="data" type="m4v:NextStartCodeType"/>
    </xsd:sequence>
</xsd:complexType>

</xsd:schema>
Listing J.2: BS Schema describing MPEG-4 Visual Simple Profile (simple types)

```
<!DOCTYPE bschema SYSTEM "entities.dtd">
<! Digital Item Adaptation ISO/IEC 21000-7 -->
<! Schema for the MPEG-4 Visual syntax -->
<! Created by Ghent University - Multimedia Lab -->
<! Modified by University of Wollongong -->

<xsd:schema xmlns:m4v="MPEG-4 Visual" xmlns="MPEG-4 Visual"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bs1x="urn:mpeg:mpeg21:2003:01-DIA-BSDL1x-NS"
xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
xmlns:bs2="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS"
elementFormDefault="qualified" attributeFormDefault="unqualified"
targetNamespace="MPEG-4 Visual">
  <xsd:annotation>
    <xsd:documentation>
      Simple Types for the MPEG-4 Visual syntax.
    </xsd:documentation>
    <xsd:appinfo>
      <bs1x:script ref="functions.js"/>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1x-NS"
schemaLocation="../../../../../../BSDL1.4.0/BSDLSchemata/BSDL1x-SoFPDAM2.xsd"/>
  <xsd:import namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
schemaLocation="../../../../../../BSDL1.4.0/BSDLSchemata/BSDL1-SoFPDAM2.xsd"/>

  <! Marker bit declaration -->
  <xsd:element name="marker_bit" type="bs1:b1" fixed="1"/>

  <! Stuffing declaration -->
  <xsd:simpleType name="Stuffing">
    <xsd:restriction base="bs1:align8"/>
  </xsd:simpleType>

  <! StartCodeType declaration -->
  <xsd:simpleType name="StartCodeType">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:annotation>
        <xsd:appinfo>
          <bs2:startCode value="000001B0"/>
          <bs2:startCode value="000001B6"/>
        </xsd:appinfo>
      </xsd:annotation>
      <xsd:length value="4"/>
    </xsd:restriction>
  </xsd:simpleType>

  <! PayloadType declaration -->
  <xsd:simpleType name="PayloadType">
    <xsd:restriction base="bs1:byteRange"/>
  </xsd:simpleType>
</xsd:schema>
```

<xsd:appinfo>
  <xsd:annotation>
    <xsd:restriction>
      <xsd:simpleType name="NextStartCodeType">
        <xsd:restriction base="xsd:base64Binary">
          <xsd:annotation>
            <xsd:appinfo>
              <bs2:startCode value="000001"/>
            </xsd:appinfo>
          </xsd:annotation>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:restriction>
  </xsd:annotation>
</xsd:appinfo>

<xsd:appinfo>
  <xsd:annotation>
    <xsd:restriction>
      <xsd:simpleType name="hex1">
        <xsd:restriction base="xsd:hexBinary">
          <xsd:length value="1"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:restriction>
  </xsd:annotation>
</xsd:appinfo>

<xsd:appinfo>
  <xsd:annotation>
    <xsd:restriction>
      <xsd:simpleType name="VOPidType">
        <xsd:restriction base="bs1:b15">
          <xsd:annotation>
            <xsd:appinfo>
              <bs2:ifUnion value="$volVersion = 1"/>
            </xsd:appinfo>
          </xsd:annotation>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:restriction>
  </xsd:annotation>
</xsd:appinfo>

<xsd:appinfo>
  <xsd:annotation>
    <xsd:restriction>
      <xsd:simpleType name="SpriteType">
        <xsd:union memberTypes="bs1:b1 bs1:b2">
          <xsd:annotation>
            <xsd:appinfo>
              <bs2:ifUnion value="$volVersion = 1"/>
            </xsd:appinfo>
          </xsd:annotation>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:restriction>
  </xsd:annotation>
</xsd:appinfo>
<bs2:ifUnion value="true()"/>
</xsd:appinfo></xsd:annotation>
</xsd:union>
</xsd:simpleType>

<!--  VOPQuantType (6.2.5) -->
<xsd:simpleType name="VOPQuantType">
  <xsd:restriction base="bs1:b9">
    <xsd:annotation><xsd:appinfo>
      <bs2:bitLength value="$m4v:quant_precision"/>
    </xsd:appinfo></xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<!--  ResyncMarkerType (6.2.5) -->
<xsd:simpleType name="ResyncMarkerType">
  <xsd:restriction base="xsd:unsignedInt">
    <xsd:annotation><xsd:appinfo>
      <bs2:bitLength value="$resyncMarkerLength"/>
    </xsd:appinfo></xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<!--  MBNumberType (6.2.5) c -->
<xsd:simpleType name="MBNumberType">
  <xsd:restriction base="xsd:unsignedInt">
    <xsd:annotation><xsd:appinfo>
      <bs2:bitLength value="$mbNumberLength"/>
    </xsd:appinfo></xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>

<!--  McbpcType (6.2.6) -->
<xsd:complexType name="McbpcType">
  <xsd:choice>
    <xsd:element name="IVOPMcbpc" type="IVOPMcbpcType" bs2:if="$m4v:vop_coding_type = &iVOP;"/>
    <xsd:element name="PVOPMcbpc" type="PVOPMcbpcType" bs2:if="$m4v:vop_coding_type = &pVOP;"/>
  </xsd:choice>
</xsd:complexType>

<!--  IVOPMcbpcType (Table B-6) -->
<xsd:complexType name="IVOPMcbpcType">
<xs...
<xsd:complexType name="PVOPMcbpcType">
  <xsd:complexContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute ref="bs1:codec" default="mcbpcPVOP.js"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<![CDATA[---]]> -- PVOPMcbpcType (Table B-7) -->
<![CDATA[---]]> -- MVDataType (Table B-15) -->
<![CDATA[---]]> -- CbpyType (Table B-15) -->

<xsd:complexType name="MVDataType">
  <xsd:complexContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute ref="bs1:codec" default="mvData.js"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<![CDATA[---]]> -- ignoring transparent blocks for now -->

<xsd:complexType name="CbpyIntraType">
  <xsd:complexContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute ref="bs1:codec" default="cbpyIntra.js"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="CbpyInterType">
  <xsd:complexContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute ref="bs1:codec" default="cbpyInter.js"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- -- MVDataType (Table B-15) --->
<!-- -- CbpyType (Table B-15) -->
<!-- -- PVOPMcbpcType (Table B-7) -->
<!-- -- *************** Table B-7 ***************** --->
<!-- -- *************** Table B-15 ***************** --->
<!-- -- *************** Table B-15 ***************** -->
<!-- MVForwardResidualType (6.2.6) -->
<xsd:simpleType name="MVForwardResidualType">
    <xsd:restriction base="bs1:b6">
        <xsd:annotation>
            <xsd:appinfo>
                <bs2:bitLength
                    value="$m4v:vop_fcode_forward − 1"/>
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:restriction>
</xsd:simpleType>

<!-- DCTdcSize (Tables B–13 and B–14) -->
<xsd:complexType name="DCTdcSizeType">
    <xsd:choice>
        <xsd:element name="DCTdcSizeLuminance" type="DCTdcSizeLuminanceType">
            <xsd:extension base="bs1:b12">
                <xsd:annotation>
                    <xsd:appinfo>
                        <bs1x:script
                            ref="dct_dc_differential.js"/>
                    </xsd:appinfo>
                </xsd:annotation>
            </xsd:extension>
        </xsd:element>
        <xsd:element name="DCTdcSizeChrominance" type="DCTdcSizeChrominanceType">
            <xsd:extension base="bs1:b12">
                <xsd:annotation>
                    <xsd:appinfo>
                        <bs1x:script
                            ref="dct_dc_differential.js"/>
                    </xsd:appinfo>
                </xsd:annotation>
            </xsd:extension>
        </xsd:element>
    </xsd:choice>
</xsd:complexType>

<!-- DCTdcSizeChrominanceType (Table B–14) -->
<xsd:complexType name="DCTdcSizeChrominanceType">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute ref="bs1:codec" default="dct_dc_chrominance.js"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- DCTdcSizeLuminanceType (Table B–13) -->
<xsd:complexType name="DCTdcSizeLuminanceType">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute ref="bs1:codec" default="dct_dc_luminance.js"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- DCTdcDifferentialType (Table B–15) -->
<xsd:simpleType name="DCTdcDifferentialType">
    <xsd:restriction base="bs1:b12">
        <xsd:annotation>
            <xsd:appinfo>
                <!-- TODO -- decode DcDifferential -->
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:restriction>
</xsd:simpleType>
<bs2:bitLength value="$m4v:dct_dc_size"/>
</xsd:appinfo></xsd:annotation>
</xsd:restriction>
</xsd:simpleType>

<!-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-->
<!-- DCTCoefficientType (7.4 Tables B–16 & B–17)  -->
<!-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-->

<xsd:complexType name="DCTCoefficientType">
<xsd:choice>
  <xsd:element name="DCTIntraCoefficient" type="DCTIntraCoefficientType"
               bs2:if="$derived_mb_type = '3' or $derived_mb_type = '4'"/>
  <xsd:element name="DCTInterCoefficient" type="DCTInterCoefficientType"
               bs2:if="true()"/>
</xsd:choice>
</xsd:complexType>

<xsd:complexType name="DCTIntraCoefficientType">
<xsd:simpleContent>
  <xsd:extension base="xsd:string">
    <xsd:attribute ref="bs1:codec" default="dct_intra_tcoeff.js"/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="DCTInterCoefficientType">
<xsd:simpleContent>
  <xsd:extension base="xsd:string">
    <xsd:attribute ref="bs1:codec" default="dct_inter_tcoeff.js"/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>

</xsd:schema>
Listing J.3: XSLT stylesheet for parser generation (main)

```xml
<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:math="http://exslt.org/math"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:bs2="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS"
    xmlns:rmc="urn:mpeg:2006:01−RMC−NS"
    xmlns:saxon="http://saxon.sf.net/" version="2.0">
    <xsl:include href="cal.xslt"/>
    <xsl:include href="functions.xslt"/>
    <xsl:include href="globals.xslt"/>
    <xsl:include href="length.xslt"/>
    <xsl:include href="linearize.xslt"/>
    <xsl:include href="actions.xslt"/>
    <xsl:include href="fsm.xslt"/>
    <xsl:include href="priorities.xslt"/>
    <xsl:variable name="preprocessedDocument" select="saxon:transform(
        saxon:compile-stylesheet(doc('preprocess.xslt')),./.)"/>
    <xsl:template match="/">
        <xsl:apply-templates select="$preprocessedDocument/*"/>
    </xsl:template>
    <xsl:variable name="targetNamespace" select="/xsd:schema/@targetNamespace"/>
    <xsl:variable name="targetPrefix" select="rmc:findPrefix($targetNamespace,./,)"/>
    <!-- imported nodes have had their name attribute augmented with a prefix -->
    <xsl:key name="types" use="resolve-QName(@name,.)">
        <xsl:apply-templates select="xsd:schema/xsd:complexType | xsd:schema/xsd:simpleType"/>
    </xsl:key>
    <!-- This won't find imported types since it doesn't look at $preprocessedDocument...and being a match statement it can't look at it.-->
    <xsl:key name="globalElements" use="resolve-QName(@name,.)">
        <xsl:apply-templates select="xsd:schema/xsd:element"/>
    </xsl:key>
    <xsl:variable name="groups" use="resolve-QName(@name,.)">
        <xsl:apply-templates select="xsd:schema/xsd:group"/>
    </xsl:variable>
    <xsl:template match="xsd:schema">
        <!-- resolve the root element -->
        <xsl:variable name="rootElementQName" select="resolve-QName(@bs2:rootElement,.)"/>
        <xsl:variable name="rootElement" select="xsd:element[QName($targetNamespace, @name)=$rootElementQName]"/>
        <xsl:variable name="rootElementType" select="if ($rootElement/@type) then rmc:localName($rootElement/@type) else concat(rmc:localName($rootElement/@name),'Type')"/>
        <xsl:variable name="readList">
            <xsl:apply-templates select="$rootElement" mode="linearize"/>
            <read name="&finalState;" again="true" testRequired="true"/>
        </xsl:variable>
    </xsl:template>
    <xsl:call-template name="actor"/>
</xsl:stylesheet>
```
<xsl:with-param name="name">BSDLParser</xsl:with-param>
<xsl:with-param name="inputs">
<xsl:call-template name="port">
  <xsl:with-param name="type">bool</xsl:with-param>
  <xsl:with-param name="name">bitstream</xsl:with-param>
</xsl:call-template>
</xsl:with-param>
<xsl:with-param name="outputs">
<xsl:call-template name="import">
  <xsl:with-param name="name" select="('caltrop','lib','BitOps')"/>
  <xsl:with-param name="kind" select="'package'"/>
</xsl:call-template>
</xsl:with-param>
<xsl:with-param name="imports">
<xsl:call-template name="import">
  <xsl:with-param name="name" select="('java','lang','Object')"/>
</xsl:call-template>
</xsl:with-param>
<xsl:with-param name="children">
  <!-- declare the bit reading action, functions etc. -->
  <xsl:call-template name="bitAction"/>
  <!-- declare the current element list -->
  <xsl:call-template name="list"/>
  <xsl:call-template name="list">
    <xsl:with-param name="name">&current;</xsl:with-param>
    <xsl:with-param name="size" select="xsd:integer(max($readList//@depth))"/>
  </xsl:call-template>
</xsl:with-param>

  <!-- create length constants -->
  <xsl:call-template name="lengthConstants"/>

  <!-- create the global variables/constants -->
  <xsl:apply-templates select="$readList" mode="globals"/>
  <xsl:with-param name="defaultPrefix" select="$targetPrefix" tunnel="yes"/>
</xsl:apply-templates>

  <!-- create the actions -->
  <xsl:apply-templates mode="buildActions" select="$readList"/>
  <xsl:with-param name="defaultPrefix" select="$targetPrefix" tunnel="yes"/>
</xsl:apply-templates>

  <!-- create the FSM -->
  <xsl:call-template name="fsm">
    <xsl:with-param name="initialState" select="concat($readList/read[1]/@name,'&existsStateSuffix;')"/>
    <xsl:with-param name="transitions"/>
  </xsl:call-template>
</xsl:call-template>
</xsl:apply-templates>
</xsl:with-param>
<!—— create the priorities ——>
<xsl:call-template name="priorities">
  <xsl:with-param name="priorities" select="$readList"/>
</xsl:call-template>

<xsl:template match="rmc:output" mode="outputPorts">
  <xsl:call-template name="port">
    <xsl:with-param name="name" select="@name"/>
  </xsl:call-template>
</xsl:template>
</xsl:stylesheet>
Listing J.4: XSLT stylesheet for parser generation (preprocess)

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="2.0">
    <xsl:include href="removeRedundantSequences.xslt"/>
    <xsl:include href="inlineComplexExtRestr.xslt"/>
    <xsl:include href="addNames.xslt"/>

    <xsl:variable name="compositeDocument" select="saxon:transform(
        saxon:compile
    −
        stylesheet(doc('includeImport.xslt')), /.)/">

    <xsl:template match="/">
        <xsl:apply−templates select="$compositeDocument/∗"/>
    </xsl:template>

    <xsl:template match="∗|@∗" mode="#all">
        <xsl:copy>
            <xsl:apply−templates select="@∗ | node()" mode="#current"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="text()" mode="#all"> <!-- strip any text --></xsl:template>
</xsl:stylesheet>
```

Listing J.5: XSLT stylesheet for parser generation (remove redundant sequences)

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="2.0">
    <!−− don't copy mandatory singular sequences −−>
    <xsl:template match="xsd:sequence[@maxOccurs='1' and not(@maxOccurs)]
        and (@minOccurs='1' and not(@minOccurs)) and
        not(parent::xsd:choice)" mode="#all">
        <xsl:apply−templates mode="#current"/>
    </xsl:template>
</xsl:stylesheet>
```

Listing J.6: XSLT stylesheet for parser generation (inline complex extensions/restrictions)

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="2.0">
    <xsl:key name="complexTypes" use="resolve−QName(@name,.)"
        match="xsd:schema/xsd:complexType"/>

    <!−− inline the base type −−>
    <xsl:template match="xsd:complexContent[xsd:extension]" mode="#all">
        <xsl:apply−templates select="key('complexTypes',
            resolve−QName(xsd:extension/@base,.))/∗" mode="#current"/>
    </xsl:template>

    <!−− TODO: harmonize constraints between base and derived type −−>
    <xsl:template match="xsd:complexContent[xsd:restriction]" mode="#all">
```
Listing J.7: XSLT stylesheets for parser generation (add names)

<xsl:stylesheet xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rmc="urn:mpeg:2006:01−RVC−NS" version="2.0">
    <xsl:template match="xsd:sequence | xsd:choice | xsd:all" mode="#all">
        <xsl:copy>
            <xsl:attribute name="name">
                <xsl:value-of select="local:name(.)"/>
                <xsl:number select="."/>
            </xsl:attribute>
            <xsl:apply-templates select="@* | node()" mode="#current"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="@ref">
        <xsl:copy/>
        <xsl:attribute name="name" select="."/>
    </xsl:template>

    <!-- anonymous types -->
    <xsl:template match="xsd:simpleType[not(@name)] | xsd:complexType[not(@name)]"
        mode="#all">
        <xsl:variable name="siblingCount"
            select="count(../*[ends-with(name(),'Type')])"/>
        <xsl:copy>
            <xsl:attribute name="name">
                <xsl:value-of select="rmc:namedAncestor(..)"/>
                <xsl:text>Type</xsl:text>
                <xsl:if test="$siblingCount > 1">
                    <xsl:number select="."/>
                </xsl:if>
                <xsl:attribute>
                    <xsl:apply-templates select="@* | node()" mode="#current"/>
                </xsl:attribute>
            </xsl:attribute>
        </xsl:copy>
    </xsl:template>

    <!-- recurse up the tree until you find a named element -->
    <xsl:function name="rmc:namedAncestor" as="xsd:string">
        <xsl:param name="typeElement" as="element()"/>
        <xsl:choose>
            <xsl:when test="$typeElement/@name">
                <xsl:value-of select="$typeElement/@name"/>
            </xsl:when>
            <xsl:when test="$typeElement/..">
                <xsl:value-of select="rmc:namedAncestor($typeElement/..)"/>
            </xsl:when>
        </xsl:choose>
    </xsl:function>
</xsl:stylesheet>
Listing J.8: XSLT stylesheet for parser generation (resolve Includes/Imports)

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:math="http://exslt.org/math"
    xmlns:rmc="urn:mpeg:2006:01−RVC−NS" version="2.0">
    <xsl:include href="functions.xslt"/>

    <xsl:template match="/@mode" mode="#all">
        <xsl:variable name="targetNamespace" select="/xsd:schema/@targetNamespace"/>
        <xsl:apply-templates mode="#current">
            <xsl:with-param name="prefix" tunnel="yes" select="rmc:findPrefix($targetNamespace,.,.)"/>
            <xsl:with-param name="namespace" tunnel="yes" select="$targetNamespace"/>
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="xsd:include" mode="#all">
        <xsl:variable name="targetNamespace" select="/xsd:schema/@targetNamespace"/>
        <xsl:variable name="includedDoc" select="document(@schemaLocation)/xsd:schema/∗"/>
        <xsl:apply-templates select="$includedDoc" mode="#current">
            <xsl:with-param name="prefix" tunnel="yes" select="rmc:findPrefix($targetNamespace,.,$includedDoc)"/>
            <xsl:with-param name="namespace" tunnel="yes" select="$targetNamespace"/>
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="xsd:import" mode="#all">
        <xsl:variable name="importedNamespace" select="@namespace"/>
        <xsl:variable name="importedDoc" select="document(@schemaLocation)/xsd:schema/∗"/>
        <xsl:apply-templates select="$importedDoc" mode="import">
            <xsl:with-param name="prefix" tunnel="yes" select="rmc:findPrefix($importedNamespace,.,$importedDoc)"/>
            <xsl:with-param name="namespace" tunnel="yes" select="$importedNamespace"/>
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="xsd:schema//@name" mode="#all">
        <xsl:param name="prefix" required="yes" tunnel="yes"/>
        <xsl:param name="namespace" required="yes" tunnel="yes"/>
        <xsl:namespace name="{$prefix}" select="$namespace"/>
        <xsl:attribute name="name" select="concat($prefix,:,.)/"/>
    </xsl:template>

    <xsl:template match="@*" mode="#all">
        <xsl:apply-templates select="@* | node()"/>
    </xsl:template>
</xsl:stylesheet>
Listing J.9: XSLT stylesheet for parser generation (functions)

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rmc="urn:mpeg:2006:01−RVC−NS" xmlns:math="http://exslt.org/math"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="2.0">
    <!-- returns the unprefixed simple name -->
    <xsl:function name="rmc:simpleLocalName" as="xsd:string?'">
        <xsl:param name="name" as="xsd:string?'"/>
        <xsl:value-of select="rmc:localName(rmc:simpleName($name))"/>
    </xsl:function>

    <!-- returns the unprefixed simple name -->
    <xsl:function name="rmc:simpleValidPrefixedName" as="xsd:string?'">
        <xsl:param name="name" as="xsd:string?'"/>
        <xsl:param name="defaultPrefix" as="xsd:string"/>
        <xsl:value-of select="rmc:validPrefixedName(rmc:simpleName($name),$defaultPrefix)"/>
    </xsl:function>

    <!-- returns the contents after the last period -->
    <xsl:function name="rmc:simpleName" as="xsd:string?'">
        <xsl:param name="name" as="xsd:string?'"/>
        <xsl:value-of select="rmc:underscore($name)"/>
    </xsl:function>

    <!-- replace : with _ -->
    <xsl:function name="rmc:validName" as="xsd:string?'">
        <xsl:param name="name" as="xsd:string?'"/>
        <xsl:value-of select="replace($name,':','')"/>
    </xsl:function>

    <!-- replace . with _ and uppercase -->
    <xsl:function name="rmc:constant" as="xsd:string?'">
        <xsl:param name="name" as="xsd:string?'"/>
        <xsl:value-of select="upper-case(rmc:underscore($name))"/>
    </xsl:function>
</xsl:stylesheet>
```
<xsl:function name="rmc:itemName" as="xsd:string?">
  <xsl:param name="stack" as="item()"/>
  <xsl:variable name="item" select="$stack[1]"/>
  <xsl:value-of>
    <xsl:if test="$stack[2]">
      <xsl:value-of select="rmc:itemName(subsequence($stack, 2))"/>
    </xsl:if>
    <xsl:choose>
      <xsl:when test="$item instance of xsd:string or $item[text() or self::text()]">
        <xsl:value-of select="$item"/>
      </xsl:when>
      <xsl:when test="$item/@name">
        <xsl:choose>
          <xsl:when test="$item/following-sibling::*[rmc:localName(@name)=rmc:localName($item/@name)] or $item/preceding-sibling::*[rmc:localName(@name)=rmc:localName($item/@name)]">
            <xsl:value-of select="rmc:validName($item/@name)"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="rmc:localName($item/@name)"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="rmc:localName($item/@name)"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:value-of>
</xsl:function>

<!-- retrieve a localname from a lexical QName -->
<xsl:function name="rmc:localName" as="xsd:string?">
  <xsl:param name="cName" as="xsd:string?"/>
  <xsl:value-of select="if (contains($cName, ':')) then substring-after($cName, ':') else $cName"/>
</xsl:function>

<!-- opposite of string--join -- zzz also known as fn:tokenize() Fix this. -->
<xsl:function name="rmc:split" as="xsd:string">
  <xsl:param name="inputString" as="xsd:string"/>
  <xsl:param name="delimiter" as="xsd:string"/>
  <xsl:sequence>
    <xsl:if (contains($inputString,$delimiter)) then substring-after($inputString,$delimiter),
    rmc:split(substring-after($inputString,$delimiter), $delimiter))
    else $inputString"/>
</xsl:function>
<xsl:function name="rmc:findPrefix" as="element()?">
  <xsl:param name="namespace" as="element()"/>
  <xsl:param name="context" as="element()"/>
  <xsl:param name="context2" as="element()"/>
  <xsl:variable name="prefix" select="($context//namespace::*[.=$namespace]/name()[string-length() &gt; 0])[1]"/>
  <xsl:variable name="prefix2" select="if ($prefix) then $prefix else ($context2//namespace::*[.=$namespace]/name()[string-length() &gt; 0])[1]"/>
  <xsl:value-of select="if ($prefix2) then $prefix2 else concat('a',round(math:random()*100) mod 100)"/>
</xsl:function>

<xsl:function name="rmc:nextReadElement" as="element()?">
  <xsl:param name="currentElement" as="element()"/>
  <xsl:choose>
    <xsl:when test="$currentElement[self::choiceOption]">
      <xsl:sequence select="$currentElement/read[1]"/>
    </xsl:when>
    <xsl:when test="$currentElement[parent::choiceOption]">
      <xsl:choose>
        <xsl:when test="$currentElement/fter=padding::read">
          <xsl:sequence select="$currentElement/fter=padding::read[1]"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:sequence select="$currentElement/fter=padding::read[1]"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence select="$currentElement/fter=padding::read[1]"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
<xsl:function name="rmc:skipPast" as="element()?">
  <xsl:param name="currentElement" as="element()"/>
  <xsl:sequence select="rmc:nextReadElement(
      $currentElement/following::skipToHere[@name=$currentElement/@name])"/>
</xsl:function>

<xsl:function name="rmc:hex2Dec" as="decimal">
  <xsl:param name="value" as="string"/>
  <xsl:return select="rmc:hex2DecInt($value,1,0)"/>
</xsl:function>

<xsl:function name="rmc:hex2DecInt" as="integer">
  <xsl:param name="value" as="string"/>
  <xsl:param name="hex-power"/>
  <xsl:param name="accum"/>
  <xsl:variable name="hex-digit" select="translate(substring($value,string-length($value),1),'abcdef','ABCDEF')"/>
  <xsl:choose>
    <xsl:when test="not(contains('0123456789ABCDEF',$hex-digit))">
      <xsl:value-of select="NaN"/>
    </xsl:when>
    <xsl:when test="string-length($remainder) = 0">
      <xsl:value-of select="$accum + $this-digit-value"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="remainder" select="substring($value,1,string-length($value)-1)"/>
      <xsl:variable name="this-digit-value" select="string-length(substring(before('0123456789ABCDEF',$hex-digit)) * $hex-power)"/>
      <xsl:choose>
        <xsl:when test="string-length($remainder) = 0">
          <xsl:value-of select="$accum + $this-digit-value"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="rmc:hex2DecInt($remainder,$hex-power + 16,$accum"/>  
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
<xsl:stylesheet>
  <xsl:otherwise/>
  <xsl:otherwise/>
  <xsl:otherwise/>
  <xsl:choose/>
  <xsl:function name="rmc:hexByteLength" as="xsd:integer">
    <xsl:param name="hexString" as="xsd:string"/>
    <xsl:value-of select="string-length($hexString) * 4"/>
  </xsl:function>
  <xsl:function name="rmc:empty" as="xsd:boolean">
    <xsl:param name="value" as="xsd:string"/>
    <xsl:sequence select="empty($value) or string-length($value) = 0"/>
  </xsl:function>
</xsl:stylesheet>
Listing J.10: XSLT stylesheet for parser generation (globals)

```xml
<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:math="http://exslt.org/math"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:rmc="urn:mpeg:2006:01−RVC−NS"
    xmlns:bs0="urn:mpeg:mpeg21:2003:01−DIA−BSDL0−NS"
    xmlns:bs2="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS"
    version="2.0">
    <xsl:template match="read[@bs0:variable='true']" mode="globals">
        <xsl:param name="defaultPrefix" tunnel="yes"/>
        <xsl:call-template name="statement">
            <xsl:with-param name="expressions">
                <xsl:call-template name="variableDeclaration">
                    <xsl:with-param name="name" select="concat('$',rmc:simpleValidPrefixedName(@name,$defaultPrefix))"/>
                </xsl:call-template>
                <xsl:call-template name="statement"></xsl:call-template>
            </xsl:with-param>
        </xsl:call-template>
        <xsl:next-match/>
    </xsl:template>
    <xsl:template match="*[@bs2:ifNext]" mode="globals">
        <xsl:variable name="ifNext" select="rmc:split(@bs2:ifNext,' ')"/>
        <xsl:call-template name="statement">
            <xsl:with-param name="expressions">
                <xsl:call-template name="variableDeclaration">
                    <xsl:with-param name="name" select="concat(rmc:constant(@name),'&ifNextLength;')"/>
                    <xsl:with-param name="initialValue" select="rmc:hexByteLength($ifNext[1])"/>
                    <xsl:with-param name="type">int</xsl:with-param>
                </xsl:call-template>
                <xsl:call-template name="statement"></xsl:call-template>
            </xsl:with-param>
        </xsl:call-template>
        <xsl:call-template name="statement">
            <xsl:with-param name="expressions">
                <xsl:call-template name="variableDeclaration">
                    <xsl:with-param name="name" select="concat(rmc:constant(@name),'&minValue;')"/>
                    <xsl:with-param name="initialValue" select="rmc:hexToDec($ifNext[1])"/>
                    <xsl:with-param name="type">int</xsl:with-param>
                </xsl:call-template>
                <xsl:call-template name="statement"></xsl:call-template>
            </xsl:with-param>
        </xsl:call-template>
        <xsl:call-template name="statement">
            <xsl:with-param name="expressions">
                <xsl:call-template name="variableDeclaration">
                    <xsl:with-param name="name" select="concat(rmc:constant(@name),'&maxValue;')"/>
                    <xsl:with-param name="initialValue" select="rmc:hexToDec($ifNext[last()])"/>
                    <xsl:with-param name="type">int</xsl:with-param>
                </xsl:call-template>
                <xsl:call-template name="statement"></xsl:call-template>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:template>
</xsl:stylesheet>
```
<idocxtype stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:rmc="urn:mpeg:2006:01−RVC−NS"

xmlns:math="http://exslt.org/math"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

version="2.0">

<xsl:template name="lengthConstants">

<xsl:variable name="context" select="."/>

<!−− create LENGTH constants for simple types −−>
<xsl:apply-templates select="/xsd:schema/xsd:simpleType" mode="length"/>

<!−− create LENGTH constants for referenced XSD types −−>
<xsl:for-each select="distinct-values(/.//xsd:element/@type)">

<xsl:variable name="typeQName" select="resolve-QName(.,$context)"/>

<xsl:if test="namespace-uri-from-QName($typeQName)=&xsdNS;">

<xsl:call-template name="statement">

<xsl:with-param name="expressions">

<xsl:call-template name="variableDeclaration">

<xsl:with-param name="name" select="concat(rmc:constant(.),&lengthSuffix;)"/>

<xsl:with-param name="type" select="int"/>

<xsl:with-param name="initialValue">

<xsl:call-template name="xsdTypeLength">

<xsl:with-param name="typeQName" select="$typeQName"/>

<xsl:call-template name="statement">

<xsl:with-param/>

<xsl:call-template>

<xsl:with-param/>

<xsl:call-template>

<xsl:if/>

<xsl:for-each />

</xsl:template>

<!−− create LENGTH constants for simple types −−>
<xsl:template match="xsd:simpleType" mode="length">

<xsl:call-template name="statement">

<xsl:with-param name="expressions">

<xsl:call-template name="variableDeclaration">

<xsl:with-param name="name" select="concat(rmc:constant(@name),&lengthSuffix;)"/>

<xsl:with-param name="type" select="int"/>

<xsl:with-param name="initialValue">

<xsl:apply-templates mode="calculateLength"/>

</xsl:template>

</xsl:template>

<!−− only need to create Length constants for anonymous inner types. −−>
<xsl:template match="xsd:simpleType[xsd:list]" mode="length">

<xsl:apply-templates select="xsd:list/*" mode="length"/>

</xsl:template></xsl:stylesheet>
Listings for Reconfigurable Media Coding

<xsl:template match="xsd:simpleType[xsd:union]" mode="length">
  <xsl:apply-templates select="xsd:union/*" mode="length"/>
</xsl:template>

<!-- TODO: test this -->
<xsl:template match="xsd:simpleType" mode="calculateLength">
  <xsl:apply-templates mode="calculateLength"/>
</xsl:template>

<xsl:template match="xsd:simpleType[xsd:list]" mode="calculateLength">
  <xsl:apply-templates select="xsd:list/*" mode="calculateLength"/>
</xsl:template>

<xsl:template match="xsd:simpleType[xsd:union]" mode="calculateLength">
  <xsl:apply-templates select="xsd:union/*" mode="calculateLength"/>
</xsl:template>

<xsl:template match="xsd:restriction" mode="calculateLength">
  <xsl:variable name="baseBitCount">
    <xsl:apply-templates select="@base" mode="calculateLength"/>
  </xsl:variable>
  <xsl:variable name="aBaseBitCount" select="if ($baseBitCount='') then 9999 else $baseBitCount"/>
  <xsl:variable name="localBitCount">
    <xsl:apply-templates select="xsd:simpleType" mode="calculateLength"/>
  </xsl:variable>
  <xsl:variable name="aLocalBitCount" select="if ($localBitCount='') then 9999 else $localBitCount"/>
  <xsl:variable name="meBitCount">
    <xsl:apply-templates select="xsd:maxExclusive" mode="calculateLength"/>
  </xsl:variable>
  <xsl:variable name="aMEBitCount" select="if ($meBitCount='') then 9999 else $meBitCount"/>
  <xsl:value-of select="math:min(($aBaseBitCount,$aLocalBitCount,$aMEBitCount))"/>
</xsl:template>

<xsl:template match="xsd:maxExclusive" mode="calculateLength">
  <xsl:value-of select="round(math:log(@value) div math:log(2))"/>
</xsl:template>

<xsl:template match="xsd:length" mode="calculateLength">
  <xsl:value-of select="@value*8"/>
</xsl:template>

<xsl:template match="@base[namespace-uri-from-QName(resolve-QName(...))=&xsdNS;]">
  <xsl:apply-templates select="key('types',resolve-QName(...))" mode="calculateLength"/>
</xsl:template>

<xsl:template match="@base" mode="calculateLength">
  <xsl:apply-templates select="key('types',resolve-QName(...))" mode="calculateLength"/>
</xsl:template>

<xsl:template match="@base[namespace-uri-from-QName(resolve-QName(...))=&xsdNS;]">
mode="calculateLength">
  <xsl:call-template name="xsdTypeLength">
    <xsl:with-param name="typeQName" select="resolve-QName(.,..)"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="xsdTypeLength">
  <xsl:param name="typeQName" />
  <xsl:choose>
    <xsl:when test="$xsdLong=$typeQName">64</xsl:when>
    <xsl:when test="$xsdInt=$typeQName">32</xsl:when>
    <xsl:when test="$xsdShort=$typeQName">16</xsl:when>
    <xsl:when test="$xsdByte=$typeQName">8</xsl:when>
    <xsl:when test="$xsdUnsignedLong=$typeQName">64</xsl:when>
    <xsl:when test="$xsdUnsignedInt=$typeQName">32</xsl:when>
    <xsl:when test="$xsdUnsignedShort=$typeQName">16</xsl:when>
    <xsl:when test="$xsdUnsignedByte=$typeQName">8</xsl:when>
  </xsl:choose>
</xsl:template>

<xsl:variable name="xsdLong" select="resolve-QName('xsd:long',/*[1])"/>
<xsl:variable name="xsdInt" select="resolve-QName('xsd:int',/*[1])"/>
<xsl:variable name="xsdShort" select="resolve-QName('xsd:short',/*[1])"/>
<xsl:variable name="xsdByte" select="resolve-QName('xsd:byte',/*[1])"/>
<xsl:variable name="xsdUnsignedLong" select="resolve-QName('xsd:unsignedLong',/*[1])"/>
<xsl:variable name="xsdUnsignedInt" select="resolve-QName('xsd:unsignedInt',/*[1])"/>
<xsl:variable name="xsdUnsignedShort" select="resolve-QName('xsd:unsignedShort',/*[1])"/>
<xsl:variable name="xsdUnsignedByte" select="resolve-QName('xsd:unsignedByte',/*[1])"/>
</xsl:stylesheet>
Listing J.12: XSLT stylesheet for parser generation (linearisation)

```xml
<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:math="http://exslt.org/math"
    xmlns:bs0="urn:mpeg:mpeg21:2003:01−DIA−BSDL0−NS"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:rmc="urn:mpeg:2006:01−RMC−NS"
    xmlns:bs2="urn:mpeg:mpeg21:2003:01−DIA−BSDL2−NS"
    version="2.0">
    <!-- create a read element for the particle -->
    <xsl:template match="xsd:element | xsd:group | xsd:sequence | xsd:all" priority="5"
        mode="linearize">
        <xsl:param name="stack" select="()"/>
        <xsl:param name="depth" select="0"/>
        <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
        <read>
            <xsl:attribute name="name" select="rmc:itemName($newStack)"/>
            <xsl:attribute name="element" select="local-name()='element'"/>
            <xsl:attribute name="depth" select="$depth"/>
            <xsl:apply-templates select="@* | *" mode="annotateRead"/>
        </read>
    </xsl:template>

    <!-- Find the next element to read — varies according to type -->
    <xsl:next-match>
        <xsl:with-param name="stack" select="$newStack"/>
        <xsl:with-param name="depth" select="$depth"/>
    </xsl:next-match>
</xsl:template>

<!-- create a read element for the choice -->
<xsl:template match="xsd:choice" mode="linearize">
    <!-- FIXME doesn't work when sequences are in choices -->
    <xsl:param name="stack" required="yes"/>
    <xsl:param name="depth" required="yes"/>
    <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
    <read choice="true">
        <xsl:attribute name="name" select="rmc:itemName($newStack)"/>
        <xsl:apply-templates select="xsd:annotation/xsd:appinfo/* | @*" mode="annotateRead"/>
        <xsl:apply-templates mode="#current">
            <xsl:with-param name="stack" select="$newStack"/>
            <xsl:with-param name="depth" select="$depth"/>
        </xsl:apply-templates>
    </read>

    <!-- possibly add a readAgain or output element -->
    <xsl:next-match>
        <xsl:with-param name="depth" select="$depth"/>
        <xsl:with-param name="stack" select="$newStack"/>
    </xsl:next-match>
</xsl:template>

<!-- wrap a choice option around everything in this particle -->
<xsl:template match="xsd:choice/*" priority="10" mode="linearize">
```

<xsl:param name="stack" required="yes"/>
<xsl:param name="depth" required="yes"/>
<choiceOption><!−− the name of the choice −−>
  <xsl:attribute name="name" select="concat(rmc:itemName($stack),',&optionName;',position())"/>
  <!−− put if and ifNext annotations on the choiceOption, rather than on the read −−>
  <xsl:apply-templates select="@* | *" mode="annotateChoiceOption"/>
  <xsl:next-match>
    <xsl:with-param name="stack" select="$stack"/>
    <xsl:with-param name="depth" select="$depth"/>
  </xsl:next-match>
</choiceOption>
</xsl:template>

<!−− process each of the children −−>
<xsl:template match="xsd:sequence | xsd:all" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
  <xsl:apply-templates mode="#current">
    <xsl:with-param name="stack" select="$newStack"/>
    <xsl:with-param name="depth" select="$depth"/>
  </xsl:apply-templates>
  <!−− possibly add a readAgain or output element −−>
  <xsl:next-match>
    <xsl:with-param name="depth" select="$depth"/>
    <xsl:with-param name="stack" select="$newStack"/>
  </xsl:next-match>
</xsl:template>

<!−− Add a readAgain element. This must occur after all other templates and children −−>
<!−− except for skipToHere and output −−>
<xsl:template match="*[@maxOccurs='unbounded' or @maxOccurs>1]" priority="−5" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <read>
    <xsl:attribute name="name" select="rmc:itemName($stack)"/>
    <xsl:attribute name="again">true</xsl:attribute>
  </read>
  <xsl:next-match>
    <xsl:with-param name="stack" select="$stack"/>
    <xsl:with-param name="depth" select="$depth"/>
  </xsl:next-match>
</xsl:template>

<!−− create a test action for an optional or multiple particle −−>
<xsl:template match="*[@minOccurs=0 or @maxOccurs='unbounded' or @maxOccurs>1]" priority="10" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
<read>
  <xsl:attribute name="isTest">true</xsl:attribute>
  <xsl:attribute name="name" select="rmc:itemName($newStack)"/>
  <xsl:apply-templates select="@* | *" mode="annotateTest"/>
  <xsl:attribute name="depth" select="$depth"/>
</read>

<xsl:next-match>
  <xsl:attribute name="name" select="rmc:itemName($newStack)"/>
  <xsl:attribute name="depth" select="$depth"/>
</xsl:next-match>
</read-template>

<!-- Add a skipToHere element. This must occur after all other templates and children except for output. -->
<xsl:template match="[@minOccurs=0 or @maxOccurs='unbounded' or @maxOccurs>1]" priority="-10" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <skipToHere>
    <xsl:attribute name="name" select="rmc:itemName($stack)"/>
  </skipToHere>
</xsl:template>

<xsl:next-match>
  <xsl:with-param name="stack" select="$stack"/>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:next-match>
</xsl:template>

<!-- add a read element to write to the output after the element is fully read. -->
<xsl:template match="xsd:element[@rmc:port]" mode="linearize" priority="-15">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <read>
    <xsl:attribute name="name" select="rmc:itemName($stack)"/>
    <xsl:attribute name="depth" select="$depth"/>
    <xsl:copy-of select="@rmc:port"/>
    <xsl:apply-templates select="@type | *" mode="annotateRead"/>
  </read>
</xsl:template>

<xsl:next-match>
  <xsl:with-param name="stack" select="$stack"/>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:next-match>
</xsl:template>

<!-- resolve the group, add the referencing element to the stack, recurse to children -->
<xsl:template match="xsd:group" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
  <xsl:variable name="group" select="key('groups',resolve-QName(@ref,))"/>
  <xsl:apply-templates select="$group/*" mode="#current"/>
  <xsl:with-param name="stack" select="$newStack"/>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:template>

<xsl:next-match>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:next-match>
</xsl:template>

<!-- possibly add a readAgain or output element -->
<xsl:template>
  <xsl:with-param name="depth" select="$depth"/>
</xsl:template>
<xsl:with-param name="stack" select="$newStack"/>
</xsl:next-match>
</xsl:template>

<!-- add the element to the stack, process the children -->
<xsl:template match="xsd:element[child::element()]" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
  <xsl:apply-templates mode="#current">
    <xsl:with-param name="stack" select="$newStack"/>
    <xsl:with-param name="depth" select="$depth"/>
  </xsl:apply-templates>
  <!-- possibly add a readAgain or output element -->
  <xsl:next-match>
    <xsl:with-param name="depth" select="$depth"/>
    <xsl:with-param name="stack" select="$newStack"/>
  </xsl:next-match>
</xsl:template>

<!-- resolve the type reference, add the element to the stack, process the referenced type -->
<xsl:template match="xsd:element[@type]" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
  <xsl:apply-templates select="key('types', resolve-QName(@type,.))" mode="#current">
    <xsl:with-param name="stack" select="$newStack"/>
    <xsl:with-param name="depth" select="$depth"/>
  </xsl:apply-templates>
  <!-- possibly add a readAgain or output element -->
  <xsl:next-match>
    <xsl:with-param name="depth" select="$depth"/>
    <xsl:with-param name="stack" select="$newStack"/>
  </xsl:next-match>
</xsl:template>

<!-- elements with a built-in type -->
<xsl:template match="xsd:element[namespace-uri-from-QName(
    resolve-QName(@type,.))=$xsdNS;]" mode="linearize">
  <xsl:param name="stack" required="yes"/>
  <xsl:param name="depth" required="yes"/>
  <!-- possibly add a readAgain or output element -->
  <xsl:next-match>
    <xsl:with-param name="depth" select="$depth"/>
    <xsl:with-param name="stack" select="$newStack"/>
  </xsl:next-match>
</xsl:template>

<!-- resolve the element reference, add the referencing element to the stack, recurse to children of referenced element -->
<xsl:template match="xsd:element[@ref]" mode="linearize">
  <xsl:param name="stack" required="yes"/>
</xsl:template>
<xsl:param name="depth" required="yes"/>
<xsl:variable name="newStack" select="if ($stack[1] is .) then $stack else (.,$stack)"/>
<xsl:variable name="referencedElt" select="key('globalElements',resolve-QName(@ref,.))"/>
<xsl:variable name="eltType" select="if ($referencedElt/@type)
then key('types',resolve-QName($referencedElt/@type,.))
else $referencedElt/∗[1]"/>
<xsl:apply-templates select="$eltType" mode="#current">
<xsl:with-param name="stack" select="$newStack"/>
<xsl:with-param name="depth" select="$depth"/>
</xsl:apply-templates>

<!−− possibly add a readAgain or output element −−>
<xsl:next-match>
<xsl:with-param name="depth" select="$depth"/>
<xsl:with-param name="stack" select="$newStack"/>
</xsl:next-match>
</xsl:template>

<!−− TODO: list type in a choice?? union type in a choice?? multiple occurrence union type??−−>
<xsl:template match="xsd:simpleType[xsd:list]" mode="linearize">
<xsl:param name="stack" required="yes"/>
<read again="true">
  <xsl:attribute name="name" select="rmc:itemName($stack)"/>
</read>
</xsl:template>

<xsl:template match="xsd:simpleType[xsd:union]" mode="linearize">
<xsl:param name="stack" required="yes"/>
<xsl:apply-templates mode="#current">
<xsl:with-param name="stack" select="$stack"/>
</xsl:apply-templates>
</xsl:template>

<xsl:template match="xsd:union" mode="linearize">
<xsl:param name="stack" required="yes"/>
<xsl:variable name="memberTypes" select="rmc:split(@memberTypes,' ')"/>
<xsl:variable name="memberTypeCount" select="count($memberTypes)"/>
<xsl:variable name="ifUnions" select="xsd:annotation/xsd:appinfo/bs2:ifUnion/@value"/>
<union>
  <xsl:attribute name="name" select="rmc:itemName($stack)"/>
  <xsl:for-each select="$memberTypes">
    <xsl:variable name="i" select="position()"/>
    <xsl:call-template name="unionType">
      <xsl:with-param name="typeName" select="."/>
      <xsl:with-param name="ifUnion" select="$ifUnions[$i]"/>
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>
</xsl:template>

<xsl:apply-templates mode="#current"/>
<xs:with-param name="ifUnion" select="subsequence($ifUnions,$memberTypeCount+1)"/>
</xs:with-param>
</xs:apply-template>
</union>
</xs:template>

<xs:template name="unionType" match="xsd:union/xsd:simpleType" mode="linearize">
  <xs:param name="ifUnion" required="no"/>
  <xs:param name="typeName" required="no" select="@name"/>
  <xs:variable name="validTypeName" select="rmc:validName($typeName)"/>
  <xs:variable name="actualIfUnion" select="if (count($ifUnion)=1) then $ifUnion[1] else $ifUnion[current()/position()]/">
    <!-- FIXME: gives incorrect result if less ifUnions than types. -->
  </xs:variable>
  <type>
    <xs:attribute name="name" select="$validTypeName"/>
    <xs:attribute name="bs2:ifUnion" select="$actualIfUnion"/>
  </type>
</xs:template>

<!-- recurse to children. -->
<xs:template match="xsd:complexType" mode="linearize">
  <xs:param name="stack" required="yes"/>
  <xs:param name="depth" required="yes"/>
  <xs:apply-templates mode="#current">
    <xs:with-param name="stack" select="$stack"/>
    <xs:with-param name="depth" select="$depth+1"/>
  </xs:apply-templates>
</xs:template>

<!-- resolve base and process it -- TODO not complete -->
<xs:template match="xsd:complexType[xsd:simpleContent]" mode="linearize">
  <xs:param name="stack" required="yes"/>
  <xs:param name="depth" required="yes"/>
  <xs:variable name="base" select="key('types',resolve-QName(xsd:simpleContent/@base,.,))"/>
  <xs:apply-templates select="$base" mode="#current">
    <xs:with-param name="stack" select="$stack"/>
    <xs:with-param name="depth" select="$depth"/>
  </xs:apply-templates>
</xs:template>

<!-- avoid having next-match call the default template -->
<xs:template match="@*" mode="linearize" priority="-1000"/>
<xs:template match="@*#" mode="annotateRead" priority="-1000"/>
<xs:template match="@*#" mode="annotateChoiceOption" priority="-1000"/>
<xs:template match="@*#" mode="annotateTest" priority="-1000"/>

<!-- annotation templates -->
<xs:template

<!-- annotate the read element with additional attributes as required -->
<xs:template
@bs2:ifNextMask | @bs2:lookAhead | @bs2:removeEmPrevByte | @bs0:variable
    mode="annotateRead"> <!-- zzz bs0 will probably become rmc -->
    <xsl:copy/>
</xsl:template>

<xsl:template match="xsd:annotation/xsd:appinfo/bs2:∗" mode="annotateRead">
    <xsl:copy/>
</xsl:template>

<xsl:template match="@bs2:ifNext | @bs2:if" mode="annotateChoiceOption">
    <xsl:copy/>
</xsl:template>

<xsl:template match="text()" mode="linearize"/>
</xsl:stylesheet>
Listing J.13: XSLT stylesheet for parser generation (FSM)

```xml
<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:math="http://exslt.org/math"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:rmc="urn:mpeg:2006:01−RMC−NS"
    version="2.0">
    <!-- read @[again] elements are used only for the next state, not directly -->
    <xsl:template match="read[@again='true']" mode="fsm" priority="1000"/>
    <!-- test transitions for an optional or multi element -->
    <xsl:template match="read[@isTest='true']" priority="10" mode="fsm">
        <xsl:variable name="next" select="rmc:nextReadElement(.)"/>
        <xsl:variable name="skipToHere" select="rmc:skipPast(.)"/>
        <xsl:call-template name="fsmTransition">
            <xsl:with-param name="current" select="."/>
            <xsl:with-param name="next" select="$skipToHere"/>
            <xsl:with-param name="currentSuffix" select="&skipActionSuffix;"/>
            <xsl:with-param name="actionSuffix" select="&existsStateSuffix;"/>
        </xsl:call-template>
        <xsl:call-template name="fsmTransition">
            <xsl:with-param name="current" select="."/>
            <xsl:with-param name="next" select="rmc:underscore(@name)"/>
            <xsl:with-param name="actionSuffix" select="&existsStateSuffix;"/>
        </xsl:call-template>
    </xsl:template>
    <!-- test transition: from this to this.exists -->
    <xsl:template match="read[@rmc:port]" priority="10" mode="fsm">
        <xsl:call-template name="fsmTransition">
            <xsl:with-param name="current" select="."/>
            <xsl:with-param name="next" select="rmc:nextReadElement(.)"/>
        </xsl:call-template>
    </xsl:template>
    <!-- from this element to the next read element -->
    <xsl:template match="read[@choice='true']" priority="5" mode="fsm">
        <xsl:apply-templates mode="fsm"/>
    </xsl:template>
</xsl:stylesheet>
```
<xsl:template match="choiceOption" mode="fsm">
  <xsl:call-template name="transition">
    <xsl:with-param name="from">
      <xsl:value-of select="rmc:underscore(parent::read/@name)"/>
      <xsl:text>&existsStateSuffix;</xsl:text>
    </xsl:with-param>
    <xsl:with-param name="to">
      <xsl:value-of select="rmc:underscore(read[1]/@name)"/>
      <xsl:text>&existsStateSuffix;</xsl:text>
    </xsl:with-param>
    <xsl:with-param name="action">
      <xsl:value-of select="read[1]/@name"/>
      <xsl:text>&testActionSuffix;</xsl:text>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:function name="rmc:nextStateSuffix" as="xsd:string?">
  <xsl:param name="next" as="element()"/>
  <xsl:choose xml:space="default">
    <xsl:when test="@rmc:port">&outputStateSuffix;</xsl:when>
    <xsl:when test="not(@isTest)">&existsStateSuffix;</xsl:when>
  </xsl:choose>
</xsl:function>
Listing J.14: XSLT stylesheet for parser generation (Actions)

<!-- ************************************************************************---
<!-- This stylesheet generates all of the actions for the actor, based on the read list -->
<!-- passed to it. -->
<!-- The general structure of the action is set by the buildActions mode, -->
<!-- and the guards and tasks are set differently depending on the various -->
<!-- attributes of the read element. -->
<!-- The final task of any action is to set readCount or readAheadCount -->
<!-- appropriately for the subsequent action. -->
--+************************************************************************---

<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet version="2.0" xmlns:math="http://exslt.org/math"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:rmc="urn:mpeg:2006:01-RVC-NS"
  xmlns:bs0="urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS"
  xmlns:bs2="urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS">
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&testActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="testGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&testActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="testGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&testActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="testGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&testActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="testGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
      </xsl:with-param>
      <xsl:with-param name="do">
        <xsl:apply-templates select="rmc:skipPast(.)" mode="setCountForNextRead"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:template>
  <xsl:template match="read[@again='true']" mode="buildActions" priority="1000"/>
  <xsl:template match="read[@isTest='true']" priority="10" mode="buildActions">
    <xsl:call-template name="action">
      <xsl:with-param name="name" select="concat(@name,'&skipActionSuffix;')"/>
      <xsl:with-param name="guard">
        <xsl:apply-templates select="." mode="skipGuard"/>
<!-- choiceOption: apply the test, and then recurse to the child. -->
<xsl:template match="choiceOption" mode="buildActions">
  <!-- Test action: set the count for the first child -->
  <xsl:call template name="action">
    <xsl:with param name="name"
      select="concat(read[1]/@name,'&testActionSuffix;')"><!-- FIXME: should eltList be storing read[1]/@name on choiceOption? -->
      <xsl:apply templates select="." mode="testGuard"/>
    </xsl:with>
    <xsl:with param name="guard">
      <xsl:apply templates select="." mode="testGuard"/>
    </xsl:with>
    <xsl:with param name="do">
      <xsl:apply templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
    </xsl:with>
  </xsl:call>
  <xsl:apply templates mode="#current"/>
</xsl:template>

<!-- FIXME: incorrectly setting setRead for following choice option. -->
</xsl:template>

<!-- Default read action -->
<xsl:template match="read" mode="buildActions">
  <xsl:call template name="action">
    <xsl:with param name="name" select="concat(@name,'&readActionSuffix;')"/>
    <xsl:with param name="guard">
      <xsl:apply templates select="." mode="readGuard"/>
    </xsl:with>
    <xsl:with param name="do">
      <xsl:apply templates select="." mode="do"/>
      <xsl:apply templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
    </xsl:with>
  </xsl:call>
  <xsl:apply templates mode="#current"/>
</xsl:template>

<!-- OUTPUT action -->
<xsl:template match="read[@rmc:port]" mode="buildActions">
  <xsl:call template name="action">
    <xsl:with param name="name" select="concat(@name,'&outputActionSuffix;')"/>
    <xsl:with param name="outputs">
      <xsl:call template name="actionPort">
        <xsl:with param name="port" select="@rmc:port"/>
        <xsl:with param name="variable">&outputVariable;</xsl:with>
        <xsl:with param name="repeat">
          <xsl:if test="@isList='true'">
            <xsl:call template name="size">
              <xsl:with param name="collection">&outputVariable;</xsl:with>
            </xsl:call>
          </xsl:if>
          <xsl:with param name="variables">
            <xsl:call template name="variableDeclaration"/>
          </xsl:with>
        </xsl:with>
      </xsl:call>
    </xsl:with>
  </xsl:call>
  <xsl:apply templates mode="#current"/>
</xsl:template>
<xsl:with-param name="name">&outputVariable;</xsl:with-param>

<xsl:with-param name="initialValue" select="rmc:currentObjects(.)"/>
</xsl:call-template>
</xsl:with-param>

<xsl:with-param name="do">
  <!-- remove the value from its parent tree -->
  <xsl:call-template name="assign">
    <xsl:with-param name="variable" select="rmc:currentObjects(.)"/>
    <xsl:with-param name="value">&null;</xsl:with-param>
  </xsl:call-template>

  <!-- set the count for the next read(Ahead) -->
  <xsl:apply-templates select="rmc:nextReadElement(.)" mode="setCountForNextRead"/>
</xsl:with-param>

<!--- ******************************************************

  -- GUARD TEMPLATES TODO: check maxOccurs value -->
  --******************************************************->

<xsl:template match="*" mode="skipGuard"/>

<!-- no guard for other skip actions -->

<!- the guard for ifNext skip action is readAheadDone() -->
<xsl:template match="*[@bs2:ifNext]" mode="skipGuard">
  <xsl:call-template name="functionCall">
    <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<!- the guard for ifNext test action is readAheadDone() and MIN_VALUE <= readAhead() and MAX_VALUE >= readAhead() -->
<xsl:template match="*[@bs2:ifNext]" mode="testGuard">
  <xsl:call-template name="operation">
    <xsl:with-param name="operation">and</xsl:with-param>
    <xsl:with-param name="operand0">
      <xsl:call-template name="operation">
        <xsl:with-param name="operation">and</xsl:with-param>
        <xsl:with-param name="operand0">
          <xsl:call-template name="functionCall">
            <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
          </xsl:call-template>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
    <xsl:with-param name="operand1">
      <xsl:call-template name="functionCall">
        <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<!- the guard for ifNext test action is readAheadDone() and MIN_VALUE <= readAhead() and MAX_VALUE >= readAhead() -->
<xsl:template match="*[@bs2:ifNext]" mode="testGuard">
  <xsl:call-template name="operation">
    <xsl:with-param name="operation">and</xsl:with-param>
    <xsl:with-param name="operand0">
      <xsl:call-template name="operation">
        <xsl:with-param name="operation">and</xsl:with-param>
        <xsl:with-param name="operand0">
          <xsl:call-template name="functionCall">
            <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
          </xsl:call-template>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
    <xsl:with-param name="operand1">
      <xsl:call-template name="functionCall">
        <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:with-param name="operand1">
  <xsl:call-template name="functionCall">
    <xsl:with-param name="name">&readAheadDone;</xsl:with-param>
  </xsl:call-template>
</xsl:with-param>
<!-- check size of current is less than the number provided -->
<xsl:template name="testMaxOccurs">
  <xsl:param name="maxOccurs"/>
  <xsl:call-template name="operation">
    <xsl:with-param name="operation"><xsl:select /></xsl:with-param>
    <xsl:with-param name="operand0" select="rmc:size(collection)"/>
    <xsl:call-template name="operation">
      <xsl:with-param name="operation">&lt;</xsl:with-param>
      <xsl:with-param name="operand0" select="$maxOccurs"/>
    </xsl:call-template>
    <xsl:with-param name="operand1" select="$maxOccurs"/>
    <xsl:call-template name="size">
      <xsl:with-param name="collection">rmc:currentObjects(.)</xsl:with-param>
    </xsl:call-template>
    <xsl:with-param name="operand1" select="$maxOccurs"/>
    <xsl:call-template name="operation">
      <xsl:with-param name="operation">&lt;</xsl:with-param>
      <xsl:with-param name="operand0" select="$maxOccurs"/>
    </xsl:call-template>
    <xsl:with-param name="operand1" select="$maxOccurs"/>
    <xsl:call-template name="size">
      <xsl:with-param name="collection">rmc:currentObjects(.)</xsl:with-param>
    </xsl:call-template>
  </xsl:call-template>
</xsl:template>

<xsl:template match="read[@type='simple']" priority="10" mode="readGuard">
  <xsl:call-template name="operation">
    <xsl:with-param name="operation">&lt;</xsl:with-param>
    <xsl:with-param name="operand1" select="$maxOccurs"/>
  </xsl:call-template>
  <xsl:call-template name="functionCall">
    <xsl:with-param name="name">&readDone;</xsl:with-param>
  </xsl:call-template>
  <xsl:with-param name="operand1" select="$maxOccurs"/>
  <xsl:call-template name="operation">
    <xsl:with-param name="operation">&lt;</xsl:with-param>
    <xsl:with-param name="operand0" select="$maxOccurs"/>
  </xsl:call-template>
  <xsl:call-template name="assign">
    <xsl:with-param name="variable">concat('$',rmc:simpleValidPrefixedName(@name,$defaultPrefix))</xsl:with-param>
    <xsl:with-param name="value">rmc:currentVariable(.)</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<!-- for elements with a variable declaration, first process the element, then make the variable assignment. -->
<xsl:template match="read[@bs0:variable='true']" mode="do" priority="10">
  <xsl:param name="defaultPrefix" tunnel="yes"/>
  <xsl:next-match/>
  <xsl:call-template name="assign">
    <xsl:with-param name="variable">concat('$',rmc:simpleValidPrefixedName(@name,$defaultPrefix))</xsl:with-param>
    <xsl:with-param name="value">rmc:currentVariable(.)</xsl:with-param>
  </xsl:call-template>
</xsl:template>
<xsl:template match="read[@depth=0]" mode="do" priority="8">
  <xsl:call-template name="assign">
    <xsl:with-param name="variable" select="rmc:currentVariable(.)"/>
    <xsl:with-param name="value"/>
    <xsl:call-template name="functionCall">
      <xsl:with-param name="name" select="rmc:localName(@typeName)"/>
    </xsl:call-template>
  </xsl:call-template>
</xsl:template>

<xsl:template match="read[@type='complex']" mode="do" priority="5">
  <xsl:call-template name="updateValueAndCurrent">
    <xsl:with-param name="value"/>
    <xsl:call-template name="functionCall">
      <xsl:with-param name="name" select="rmc:localName(@typeName)"/>
    </xsl:call-template>
  </xsl:call-template>
</xsl:template>

<xsl:template match="read[@type='simple']" mode="do">
  <xsl:call-template name="updateValueAndCurrent">
    <xsl:with-param name="value" select="&readResult;"/>
    <xsl:call-template name="functionCall">
      <xsl:with-param name="name" select="rmc:localName(@typeName)"/>
    </xsl:call-template>
  </xsl:call-template>
</xsl:template>

<xsl:template name="updateValueAndCurrent">
  <xsl:call-template name="assign">
    <xsl:with-param name="variable" select="rmc:currentObject(.,true())"/>
    <xsl:with-param name="value" select="$value"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="current[n-1].child = childType() or readResult(); //update the value
current[n] = current[n-1].child; //update the current pointer">
  <xsl:param name="updateValueAndCurrent"/>
  <xsl:call-template name="assign">
    <xsl:with-param name="variable" select="rmc:currentObject(.,true())"/>
    <xsl:with-param name="value" select="$value"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="rmc:currentVariable">
  <xsl:param name="context"/>
</xsl:template>
<xsl:text>&amp;current;</xsl:text>
<xsl:call-template name="index">
  <xsl:with-param name="index" select="$context/@depth"/>
</xsl:call-template>
</xsl:function>

<xsl:function name="rmc:currentObject">
  <xsl:param name="context"/>
  <xsl:param name="appending"/>

  <xsl:value-of select="rmc:currentObjects($context)="/n>
  <xsl:if test="$context/@isList = 'true'">
    <xsl:call-template name="index">
      <xsl:with-param name="index" select="$context/@depth - 1"/>
    </xsl:call-template>
    <xsl:if test="not($appending)">
      -1</xsl:if>
    <xsl:if test="not($appending)">
      return the last actual value instead of the first empty slot -->
    </xsl:if>
  </xsl:if>
</xsl:function>

<!-- current[n].element //if element is a multi-object this gives you the whole list -->
<xsl:function name="rmc:currentObjects">
  <xsl:param name="context"/>
  <xsl:call-template name="dereference">
    <xsl:with-param name="parent" select="rmc:simpleName($context/@name)="/n>
    <xsl:call-template>
      <xsl:with-param name="index" select="$context/@depth - 1"/>
    </xsl:call-template>
    <xsl:with-param name="child" select="rmc:currentObject($context)="/n>
  </xsl:call-template>
</xsl:function>

<!-- ******************************************
-->
<!-- Note: The context item for all of these templates is the following read -->
<!-- element of the one for which the action is being built. -->
<!-- ******************************************
-->
<!-- resolve nextReadElement, apply setCountForNextRead templates -->
<xsl:template name="setCountForNextRead">
  <xsl:variable name="next" select="rmc:nextReadElement(")="/n>
  <xsl:apply-templates select="$next" mode="setCountForNextRead"/>
</xsl:template>

<!-- next element has an ifNext. Set readAhead to the ifNext length -->
<xsl:template match="read[@bs2:ifNext] priority="10" mode="setCountForNextRead"">
&lt;xsl:template name="statement"&gt;
  &lt;xsl:with-param name="expressions"&gt;
    &lt;xsl:call-template name="functionCall"&gt;
      &lt;xsl:with-param name="name"&gt;&amp;setReadAhead;&lt;/xsl:with-param&gt;
      &lt;xsl:with-param name="arguments"&gt;
        &lt;xsl:value-of select="rmc:constant(@name)"/&gt;
      &lt;/xsl:with-param&gt;
    &lt;/xsl:call-template&gt;
  &lt;/xsl:with-param&gt;
&lt;/xsl:template&gt;

&lt;xsl:template match="read[@choice='true']" priority="5" mode="setCountForNextRead"&gt;
  &lt;xsl:variable name="ifNexts" select="choiceOption[bs2:ifNext]"/&gt;&lt;!-- TODO: should refactor this to a constant called CHOICE_READAHEAD --&gt;
  &lt;xsl:if test="not(empty($ifNexts))"&gt;
    &lt;xsl:variable name="maxIfNextLength" select="rmc:maxIfNextLength($ifNexts)"/&gt;
    &lt;xsl:call-template name="statement"&gt;
      &lt;xsl:with-param name="expressions"&gt;
        &lt;xsl:call-template name="functionCall"&gt;
          &lt;xsl:with-param name="name"&gt;&amp;setReadAhead;&lt;/xsl:with-param&gt;
          &lt;xsl:with-param name="arguments"&gt;
            &lt;xsl:value-of select="rmc:constant($maxIfNextLength/@name)"/&gt;
          &lt;/xsl:with-param&gt;
        &lt;/xsl:call-template&gt;
      &lt;/xsl:with-param&gt;
    &lt;/xsl:call-template&gt;
  &lt;/xsl:if&gt;
&lt;/xsl:template&gt;

&lt;xsl:function name="rmc:maxIfNextLength" as="element()?"&gt;
  &lt;xsl:param name="ifNexts"/&gt;
  &lt;xsl:if test="not(empty($ifNexts))"&gt;
    &lt;xsl:variable name="ifNext1" select="rmc:split($ifNexts[1]/bs2:ifNext,' ')[1]"/&gt;
    &lt;xsl:variable name="maxIfNext" select="rmc:maxIfNextLength(subsequence($ifNexts,2))"/&gt;
    &lt;xsl:variable name="ifNext2" select="rmc:split($maxIfNext/@bs2:ifNext,' ')[1]"/&gt;
    &lt;xsl:choose&gt;
      &lt;xsl:when test="empty($maxIfNext) or rmc:hexByteLength($ifNext1) &gt; rmc:hexByteLength($ifNext2)"&gt;
        &lt;xsl:sequence select="$ifNexts[1]"/&gt;
      &lt;/xsl:when&gt;
      &lt;xsl:otherwise&gt;
        &lt;xsl:sequence select="$maxIfNext"/&gt;
      &lt;/xsl:otherwise&gt;
    &lt;/xsl:choose&gt;
  &lt;/xsl:if&gt;
&lt;/xsl:function&gt;
<!−− nothing to do. −−>
<xsl:template match="read[@rmc:port or @type='complex' or @name='&finalState;']"
         priority="2" mode="setCountForNextRead"/>

<!−− next element is a simple read. set the read value to the bit length. −−>
<xsl:template match="read[@type='simple']" priority="1"
         mode="setCountForNextRead">
    <xsl:call-template name="statement">
        <xsl:with-param name="expressions">
            <xsl:call-template name="functionCall">
                <xsl:with-param name="name">&setRead;</xsl:with-param>
                <xsl:with-param name="arguments" select="concat(rmc:constant(@typeName),&lengthSuffix;);"/>
            </xsl:call-template>
        </xsl:with-param>
    </xsl:call-template>
</xsl:template>

Listing J.15: XSLT stylesheet for parser generation (Priorities)
Listing J.16: XSLT stylesheet for parser generation (CAL templates)

```xml
<!DOCTYPE stylesheet SYSTEM "entities.dtd">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:rmc="urn:mpeg:2006:01–RVC–NS" version="2.0">
    <xsl:output method="text" omit-xml-declaration="yes"/>

    <xsl:variable name="setFunc" select="'setBitCount'"/>
    <xsl:variable name="resultFunc" select="'readResult'"/>
    <xsl:variable name="completeFunc" select="'readComplete'"/>
    <xsl:variable name="commaNewline">&nl;&tab;&tab;</xsl:variable>

    <xsl:template name="actor">
        <xsl:param name="name" required="yes"/>
        <xsl:param name="children"/>
        <xsl:param name="inputs"/>
        <xsl:param name="outputs"/>
        <xsl:param name="imports"/>
        <xsl:param name="parameters" as="node()"/>
        <xsl:value-of select="$imports"/>
        <xsl:text>&nl;actor (</xsl:text>
        <xsl:value-of select="string-join($parameters/parameter,', ')"/>
        <xsl:text>)</xsl:text>
        <xsl:text>==</xsl:text>
        <xsl:value-of select="string-join($inputs/item,', ')"/>
        <xsl:text>:&nl;</xsl:text>
        <xsl:value-of select="string-join($outputs/item,', ')"/>
        <xsl:text>&nl;end</xsl:text>
    </xsl:template>

    <xsl:template name="import">
        <!-- should be a sequence with each package in a slot -->
        <xsl:param name="name" required="yes"/>
        <xsl:param name="kind" select="'single'"/>
        <xsl:text>import </xsl:text>
        <xsl:if test="$kind='package'">all </xsl:if>
        <xsl:value-of select="string-join($name,'.')"/>
    </xsl:template>

    <xsl:template name="port">
        <xsl:param name="name" required="yes"/>
        <xsl:param name="type" required="no"/>
        <item>
            <xsl:if test="not(rmc:empty($type))">
                <xsl:value-of select="$type"/>
                <xsl:text> </xsl:text>
            </xsl:if>
            <xsl:value-of select="$name"/>
        </item>
    </xsl:template>
</xsl:stylesheet>
```
// Utility action to read a specified number of bits.
// This is an unnamed action, i.e. it is always enabled and has highest priority.
// Use the procedures set_bits_to_read() to start reading, test for
// completion with the boolean done_reading_bits() and get the value
// with read_result(). Use the done function in a guard to wait for the
// reading to be complete.
int(size=7) bits_to_read_count := -1;
int(size=33) read_result_in_progress;
procedure set_bits_to_read( int count )
begin
  bits_to_read_count := count - 1;
  read_result_in_progress := 0;
end
function done_reading_bits() --&gt; bool : bits_to_read_count &lt; 0 end
function read_result() --&gt; int : read_result_in_progress end
action bits:[ b ] =&gt;
  guard
    not done_reading_bits()
do
    read_result_in_progress := bitor( lshift( read_result_in_progress, 1), if b then 1 else 0 end);
    bits_to_read_count := bits_to_read_count - 1;
end
</xsl:template>

<xsl:template name="action">
  <xsl:param name="name" required="no"/>
  <xsl:param name="inputs" as="node()" required="no"/>
  <xsl:param name="outputs" as="node()" required="no"/>
  <xsl:param name="guard" required="no"/>
  <xsl:param name="variables" required="no"/>
  <xsl:param name="do" required="no"/>
  <xsl:text>&lt;xsl:value-of select="$name"/&gt;</xsl:text>
  <xsl:if test="not(rmc:empty($guard))">
    <xsl:text>&lt;xsl:value-of select="string-join($outputs/item, ', ')"/&gt;</xsl:text>
  </xsl:if>
  <xsl:if test="not(rmc:empty($variables))">
    <xsl:text>&lt;xsl:value-of select="string-join($inputs/item, ', ')"/&gt;</xsl:text>
  </xsl:if>
</xsl:template>
<xsl:template name="assign">
  <xsl:param name="variable" required="yes"/>
  <xsl:param name="value" required="yes"/>
  <xsl:text>&tab;&tab;</xsl:text>
  <xsl:value-of select="$variable"/>
  <xsl:text>=</xsl:text>
  <xsl:value-of select="$value"/>
  <xsl:text>;&nl;</xsl:text>
</xsl:template>

<xsl:template name="index">
  ![−− zzz workaround for cal interpreter bug − remove [] −− ]
  <xsl:param name="index" required="yes"/>
  <xsl:text>[
</xsl:text>
  <xsl:value-of select="$index"/>
  <xsl:text>]
</xsl:template>

<xsl:template name="size">
  <xsl:param name="collection" required="yes"/>
  <xsl:text>#
</xsl:text>
  <xsl:value-of select="$collection"/>
</xsl:template>

<xsl:template name="dereference">
  <xsl:param name="parent" required="yes"/>
  <xsl:param name="child" required="yes"/>
  <xsl:value-of select="$parent"/>
  <xsl:text>.</xsl:text>
  <xsl:value-of select="$child"/>
</xsl:template>

<xsl:template name="statement">
  <xsl:param name="expressions" required="no"/>
  <xsl:if test="not(rmc:empty($expressions))">
    <xsl:text>&tab;&tab;</xsl:text>
    <xsl:value-of select="$expressions"/>
    <xsl:text>;&nl;</xsl:text>
  </xsl:if>
</xsl:template>

<xsl:template name="list1">
  <xsl:param name="name" as="xsd:string" required="yes"/>
  <xsl:param name="size" required="no"/>
  <xsl:param name="type" required="no"/>
  <xsl:param name="initialValue" as="xsd:string" required="no"/>
  <xsl:variable name="sizePresent" select="not(rmc:empty($size))"/>
  <xsl:variable name="typePresent" select="not(rmc:empty($type))"/>
  <xsl:variable name="nullValue" select="if ($type='int') then 0 else 'null'"/>
  <item>
<xsl:template name="list">
  <xsl:if test="$size or $typePresent">
    type:<xsl:text>
    <xsl:value-of select="$type"/>
    </xsl:text>
  </xsl:if>
  <xsl:if test="$sizePresent and $typePresent">
    size:<xsl:text>
    <xsl:value-of select="$size"/>
    </xsl:text>
  </xsl:if>
  <xsl:choose>
    <xsl:when test="not(rmc:empty($initialValue))">
      <xsl:text>:= 
      <xsl:value-of select="string-join($initialValue, ', ')' />
      </xsl:text>
    </xsl:when>
    <xsl:when test="$sizePresent">
      <xsl:variable name="string" select="for $x in 1 to $size return $nullValue"/>
      <xsl:text>:= 
      <xsl:value-of select="string-join($string, ', ')' />
      </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:choose>
        <xsl:when test="not(rmc:empty($initialValue))">
          <xsl:choose>
            <xsl:when test="not(rmc:empty($initialValue))">
              <xsl:choose>
                <xsl:when test="not(rmc:empty($initialValue))">
                  <xsl:choose>
                    <xsl:when test="not(rmc:empty($initialValue))">
                      <xsl:choose>
                        <xsl:when test="not(rmc:empty($initialValue))">
                          <xsl:choose>
                            <xsl:when test="not(rmc:empty($initialValue))">
                              <xsl:choose>
                                <xsl:when test="not(rmc:empty($initialValue))">
                                  <xsl:choose>
                                    <xsl:when test="not(rmc:empty($initialValue))">
                                      <xsl:choose>
                                        <xsl:when test="not(rmc:empty($initialValue))">
                                          <xsl:choose>
                                            <xsl:when test="not(rmc:empty($initialValue))">
                                              <xsl:choose>
                                                <xsl:when test="not(rmc:empty($initialValue))">
                                                  <xsl:choose>
                                                    <xsl:when test="not(rmc:empty($initialValue))">
                                                      <xsl:choose>
                                                        <xsl:when test="not(rmc:empty($initialValue))">
                                                          <xsl:choose>
                                                            <xsl:when test="not(rmc:empty($initialValue))">
                                                              <xsl:choose>
                                                                <xsl:when test="not(rmc:empty($initialValue))">
                                                                  <xsl:choose>
                                                                    <xsl:when test="not(rmc:empty($initialValue))">
                                                                      <xsl:choose>
                                                                        <xsl:when test="not(rmc:empty($initialValue))">
                                                                          <xsl:choose>
                                                                            <xsl:when test="not(rmc:empty($initialValue))">
                                                                              <xsl:choose>
                                                                                <xsl:when test="not(rmc:empty($initialValue))">
                                                                                  <xsl:choose>
                                                                                    <xsl:when test="not(rmc:empty($initialValue))">
                                                                                      <xsl:choose>
                                                                                      <xsl:when test="not(rmc:empty($initialValue))">
                                                                                      </xsl:choose>
                                                                                      </xsl:when>
                                                                                      </xsl:otherwise>
                                                                                      </xsl:otherwise>
                                                                               </xsl:otherwise>
                                                                              </xsl:otherwise>
                                                                            </xsl:otherwise>
                                                                          </xsl:otherwise>
                                                                        </xsl:otherwise>
                                                                      </xsl:otherwise>
                                                                    </xsl:otherwise>
                                                                  </xsl:otherwise>
                                                                </xsl:otherwise>
                                                              </xsl:otherwise>
                                                            </xsl:otherwise>
                                                          </xsl:otherwise>
                                                        </xsl:otherwise>
                                                      </xsl:otherwise>
                                                    </xsl:otherwise>
                                                  </xsl:otherwise>
                                                </xsl:otherwise>
                                              </xsl:otherwise>
                                            </xsl:otherwise>
                                          </xsl:otherwise>
                                        </xsl:otherwise>
                                      </xsl:otherwise>
                                    </xsl:otherwise>
                                  </xsl:otherwise>
                                </xsl:otherwise>
                              </xsl:otherwise>
                            </xsl:otherwise>
                          </xsl:otherwise>
                        </xsl:otherwise>
                      </xsl:otherwise>
                    </xsl:otherwise>
                  </xsl:otherwise>
                </xsl:otherwise>
              </xsl:otherwise>
            </xsl:otherwise>
          </xsl:otherwise>
        </xsl:otherwise>
      </xsl:otherwise>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:param name="name" required="yes"/>
<xsl:param name="arguments" required="no"/>
<xsl:value-of select="$name"/>
<xsl:text>
<xsl:value-of select="string-join($arguments,',')"/>
</xsl:text>
</xsl:template>

<-- for convenience -->
<xsl:template name="increment">
    <xsl:param name="variable" required="yes"/>
    <xsl:call-template name="assign">
        <xsl:with-param name="variable" select="$variable"/>
        <xsl:with-param name="value">
            <xsl:call-template name="operation">
                <xsl:with-param name="operand0" select="$variable"/>
                <xsl:with-param name="operation">+</xsl:with-param>
                <xsl:with-param name="operand1">1</xsl:with-param>
                <xsl:call-template>
                    <xsl:with-param name="variable" select="$variable"/>
                </xsl:call-template>
            </xsl:call-template>
        </xsl:with-param>
    </xsl:call-template>
</xsl:template>

<xsl:template name="operation">
    <xsl:param name="operation" required="yes"/>
    <xsl:param name="operand0" required="yes"/>
    <xsl:param name="operand1" required="no"/>
    <xsl:value-of select="$operand0"/>
    <xsl:if test="not(rmc:empty($operand1))"> <!-- makes it simpler to assemble guards -->
        <xsl:text>
        <xsl:value-of select="$operation"/>
        <xsl:text>
        <xsl:value-of select="$operand1"/>
        </xsl:if>
    </xsl:template>

<xsl:template name="transition">
    <xsl:param name="from" required="yes"/>
    <xsl:param name="to" required="yes"/>
    <xsl:param name="action" required="yes"/>
    <item>
        <xsl:text>&amp;tab;&amp;tab;</xsl:text>
        <xsl:value-of select="string-join($from,'')"/>
        <xsl:text>&amp;tab;</xsl:text>
        <xsl:value-of select="string-join($action,'.')"/>
        <xsl:text>&amp;tab;</xsl:text>
        <xsl:value-of select="string-join($to,'')"/>
        <xsl:text>&amp;nl;</xsl:text>
    </item>
</xsl:template>

<xsl:template name="priorities">
    <xsl:param name="priorities" as="node()" required="yes"/>
</xsl:template>
<xsl:template name="priority">
<xsl:param name="greater" required="yes"/>
<xsl:param name="lesser" required="yes"/>

<xsl:value-of select="$greater"/>
<xsl:text>&gt;&lt;/xsl:text>
<xsl:value-of select="$lesser"/>
<xsl:text>&lt;/xsl:text>
</item>
</xsl:template>

<xsl:template name="foreach">
<xsl:param name="variable" required="yes"/>
<xsl:param name="collection" required="yes"/>
<xsl:param name="statements"/>

<xsl:text>&lt;/xsl:text>
<xsl:value-of select="$variable"/>
<xsl:text> in </xsl:text>
<xsl:value-of select="$collection"/>
<xsl:text> do &nl;&lt;/xsl:text>
<xsl:value-of select="$statements"/>
<xsl:text>&lt;/xsl:text>
</xsl:template>
</xsl:stylesheet>