Beschrijvingsgedreven aanpassing van mediabronnen

Description-Driven Adaptation of Media Resources

Wesley De Neve

Promotor: prof. dr. ir. R. Van de Walle
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2006 - 2007

UNIVERSITEIT
GENT

ISBN 978-90-8578-132-5
NUR 980, 986
Wettelijk depot: D/2007/10.500/6

Dankwoord

Met het indienen van dit proefschrift sluit ik een periode van ruim vier jaar af. In deze tijdspanne heb ik de mogelijkheid gekregen om in een boeiend domein aan wetenschappelijk onderzoek te doen. Dit is dan ook een uitgelezen kans om de mensen te bedanken zonder wie dit werk niet mogelijk zou zijn geweest.

In de eerste plaats wil ik mijn promotor, prof. dr. ir. Rik Van de Walle bedanken. Dankzij zijn onuitputtelijke inzet is het Multimedia Lab als onderzoeksgroep uitgegroeid tot een gevestigde waarde op nationaal en internationaal vlak. Deze omgeving vormde voor mij dan ook het ideale kader voor het behalen van mijn doctoraat. Verder wil ik mijn promotor eveneens bedanken voor de kansen die hij mij geboden heeft om mijn onderzoek voor te stellen op verschillende internationale conferenties. Het bijwonen van deze conferenties heeft me als onderzoeker, maar ook als mens, enorm veel bijgebracht.

Ook mijn vroegere (Boris Rogge, Sam Lerouge) en huidige collega's van het Multimedia Lab wil ik uitgebreid bedanken. Het was voor mij erg aangenaam in een dergelijke open en vriendschappelijke omgeving te kunnen werken. Een aantal collega's hebben grote delen van dit proefschrift heel grondig nagelezen. Hun opmerkingen en suggesties hebben de kwaliteit van dit werk in grote mate verhoogd. Ik wil hen dan ook in het bijzonder danken voor deze inspanningen. Ook Rita Breems en Ellen Lammens bedank ik van harte voor de administratieve ondersteuning en de leuke babbels tijdens het middageten.

Een bijzonder woord van dank gaat ook uit naar Davy Van Deursen voor de bijdrage die hij, als thesisstudent en als collega, tot dit werk heeft geleverd. Ook Peter Lambert en Davy De Schrijver wens ik te bedanken voor de talrijke en boeiende discussies over H.264/AVC en MPEG-21 BSDL. In deze context zou ik eveneens Gary J. Sullivan (Microsoft Corporation) en Miska M. Hannuksela (Nokia) willen danken, en dit voor de tips en de informatie die zij verstrekt hebben met betrekking tot het gebruik van sub-sequences in de H.264/AVC-standaard. Tevens wens ik Danny Hong (Columbia University) te bedanken voor het delen van zijn kennis met betrekking tot de werking van XFlavor. Ook de anonieme reviewers van het door Elsevier gepubliceerde

manuscript over BFlavor wil ik uitdrukkelijk bedanken voor de uitgebreide en opbouwende feedback.

Ik wens ook mijn ouders te bedanken voor de vele kansen die zij mij gaven, alsook voor hun raad, steun en vertrouwen in mij. Verder wil ik ook mijn zus Lindsey bedanken voor haar relativerende kijk op haar grote broer. Ook wens ik al mijn vrienden van harte te bedanken voor de vele boeiende en leuke momenten, zowel in de online als offline wereld. Zij zorgden ervoor dat ik regelmatig mijn zinnen kon verzetten en dat de vele uren voor het computerscherm draagbaar bleven.

Tot slot wil ik nog een aantal bijzondere mensen bedanken die ik heb leren kennen dankzij mijn onderzoek, zij het in een meer informele context: Charles-Frederik Hollemeersch, Elisabeth Hollemeersch, en Mieke De Ketelaere, voor de onvergetelijke deelname aan de Imagine Cup in Brazilië; R.A. Skip Campbell, voor de avontuurlijke roadtrip in Coloroda; Leyda en Florence (Xiafen Zhang), voor jullie boeiende verhalen over China; en Sehee Lee, voor het gidsen in het Land van de Ochtendkalmte.

Wesley M. De Neve
31 Augustus 2006

Summary

The last decade has witnessed a significant number of innovative developments in the multimedia ecosystem. Advanced media formats have emerged for the efficient representation, storage, and presentation of digital media resources. New network technologies have been devised, wired and wireless, providing access to audio-visual information services such as online music stores, movie download services, and video blogs. A plethora of networked mobile devices has popped up as well, ranging from cell phones to personal entertainment devices, often having sufficient processing power for the playback of multimedia presentations. From these observations, it is clear that the multimedia landscape is characterized by a vast diversity in terms of media formats, network capabilities, and device properties.

The ever-increasing heterogeneity in the multimedia consumption chain poses a number of new challenges. One such challenge is the realization of the *Universal Multimedia Access* paradigm, which is the notion that multimedia content should be accessible at any place, at any time, and with any device. As acknowledged by the Moving Picture Experts Group (MPEG), the successful realization of ubiquitous and seamless access to multimedia content requires an appropriate reaction from different knowledge domains.

- The answer of MPEG's coding community consists of the specification of scalable or layered coding schemes. Indeed, the picture rate and spatial resolution of scalable video resources can for instance be adapted in a straightforward way to meet the different constraints that are imposed by a particular usage context (e.g., constraints in terms of available bandwidth, screen resolution, and so on).

- The answer of MPEG's metadata community consists of the development of a number of description tools. These tools are for example used to describe the properties of media resources and the capabilities or constraints of usage environments. The resulting descriptions enable the construction of a format-agnostic content adaptation system that is able

> to maximize the user experience, for the consumption of a particular multimedia presentation in a well-defined usage environment.

In this dissertation, we have first studied a number of concepts and design principles of the state-of-the-art H.264/MPEG-4 Advanced Video Coding standard, typically abbreviated as H.264/AVC. The H.264/AVC specification incorporates the latest advances in standardized video coding technology. As demonstrated by our experiments, as well as by other scientific and technical sources, H.264/AVC provides up to 50% bit rate savings for equivalent perceptual quality relative to the performance of prior video coding standards.

The design of the H.264/AVC standard is, besides efficiency, also characterized by a flexibility for use over a broad variety of network types and application domains[1]. In this context, we have reviewed several content adaptation tools that are part of the initial version of the H.264/AVC specification. Examples include the exploitation of multi-layered temporal scalability and the use of Flexible Macroblock Ordering (FMO) for region of interest coding.

The emphasis of our review was put on providing a complete and detailed picture regarding H.264/AVC's temporal adaptivity provisions. As such, our overview contains an extensive discussion with respect to the use of sub-sequences and sub-sequence layers, coding patterns based on hierarchical bidirectionally coded pictures (B pictures), and Supplemental Enhancement Information messages (SEI messages) for communicating the bitstream structure to a bitstream extractor or decoder. Sub-sequences are employed to constrain H.264/AVC's coding flexibility in a minimal way. This allows the execution of meaningful adaptations in the temporal dimension.

More powerful adaptivity features and SEI messages are incorporated in a newly developed amendment to the H.264/AVC specification, which is commonly referred to as H.264/AVC Scalable Video Coding (SVC). This amendment includes explicit support for spatial and quality scalability; temporal adaptivity tools are inherited from the first version of the H.264/AVC standard.

Further, the principles of Bitstream Syntax Description-driven (BSD-driven) content adaptation were also discussed in this dissertation. A BSD contains a description of the high-level structure of a binary media resource, typically expressed using the eXtensible Markup Language (XML). This XML-based description, i.e. the BSD, can be transformed to reflect a desired adaptation of the binary media resource. The transformed BSD can subsequently be used to automatically create an adapted media resource by relying on a format-independent content adaptation engine. This adapted media resource is then suited for consumption in a particular usage environment.

[1]Efficient and flexible are two terms that are often used in the context of H.264/AVC.

Two different approaches for BSD-driven content adaptation were studied in more detail: a standardized framework driven by the MPEG-21 Bitstream Syntax Description Language (MPEG-21 BSDL) and a framework based on the use of the Formal Language for Audio-Visual Object Representation, extended with XML features (XFlavor). Both technologies provide different means for the automatic translation of the structure of a binary media resource into an XML-based BSD, and for the subsequent generation of a tailored bitstream using a transformed BSD.

The high-level structure of a number of common video coding and container formats was described using MPEG-21 BSDL and XFlavor. Particular attention was paid to the construction of a description in MPEG-21 BSDL for the first version of H.264/AVC, as this effort exposed a few shortcomings in the schema language in question, requiring the development of a number of non-normative extensions to MPEG-21 BSDL.

Besides testing the expressive power of MPEG-21 BSDL and XFlavor, we also evaluated their performance in the context of the different media formats described, targeting applications such as BSD-driven temporal adaptation and demultiplexing. Our experiments resulted in the identification of several performance bottlenecks, in particular the slow and memory-consuming generation of BSDs using BSDL's BintoBSD Parser (which allows the format-agnostic and automatic generation of BSDs), the verbose BSDs produced by XFlavor-based parsers, and the memory-consuming transformation of BSDs using eXtensible Stylesheet Language Transformations (XSLT).

The performance issues of BSDL's BintoBSD Parser are due to the storage of an entire BSD in the system memory, needed to correctly steer the processing behavior of this parser. To enable a more efficient generation of BSDs, we have proposed BFlavor (BSDL + XFlavor), a new description tool that is the result of a cross-fertilization between MPEG-21 BSDL and XFlavor. Indeed, BFlavor harmonizes BSDL and XFlavor by combining their strengths and by eliminating their weaknesses. In particular, the processing efficiency and expressive power of XFlavor, together with the ability of BSDL to create high-level BSDs, were our key motives for its development. As such, the use of BFlavor-based BSD producers, which are format-specific but generated automatically by a format-independent process, is an efficient alternative to the use of BSDL's format-neutral but inefficient BintoBSD Parser. The development of BFlavor can be considered the main contribution of this research.

The expressive power and performance of a hybrid, BFlavor-driven content adaptation chain, compared to tool chains entirely based on either BSDL or XFlavor, were illustrated by several experiments. One series of experiments particularly targeted the exploitation of multi-layered temporal scalability in

H.264/AVC, paying special attention to the combined use of sub-sequences and SEI messages. BFlavor was the only tool to offer an elegant and practical solution for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain.

In this dissertation, we have also outlined the BSD-based construction of placeholder slices and pictures for a number of video coding formats. These artificial syntax structures allow to eliminate a number of unwanted side-effects, resulting from a BSD-driven content adaptation step in the compressed domain. The use of placeholder slices and pictures was discussed in more detail in the context of the BSD-based exploitation of temporal and Region Of Interest (ROI) scalability in the first edition of the H.264/AVC standard, respectively providing a solution for synchronization and conformance issues.

A final contribution consists of introducing a real-time work flow for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain. The key technologies used were BFlavor for the efficient generation of BSDs, Streaming Transformations for XML (STX) for the efficient transformation of BSDs, and BSDL's format-neutral BSDtoBin Parser for the efficient construction of tailored video bitstreams. Extensive performance data were provided for several use cases, involving the exploitation of temporal scalability by dropping slices, the enhanced exploitation of temporal scalability by relying on placeholder slices, and the creation of video skims (i.e., video summaries). The latter application is made possible by enriching a BSD with additional metadata to steer the BSD transformation process.

To conclude, we hope we have convinced the reader that this dissertation, although limited in its scope, contributed to bridging the gap between content and context, supporting the vision that in the end, the user, and not the terminal and the network, is to be considered the real point of attention in the multimedia consumption chain.

Samenvatting

De laatste jaren zijn we getuige geweest van een aantal innovatieve ontwikkelingen in het multimedia-ecosysteem. Geavanceerde mediaformaten zijn beschikbaar voor de efficiënte representatie, opslag, en presentatie van digitale mediabronnen. Nieuwe netwerktechnologieën werden ontwikkeld, bedraad en draadloos, die toegang verlenen tot audiovisuele diensten zoals online muziekwinkels, online filmverkoop en videogedreven dagboeken op het Internet. Een waaier aan mobiele toestellen, gaande van mobieltjes tot draagbare mediaspelers, is eveneens in staat om multimediapresentaties op een vlotte manier weer te geven.

De voorgaande vaststellingen maken duidelijk dat het multimedialandschap gekenmerkt wordt door een enorme diversiteit in termen van beschikbare mediaformaten, netwerkmogelijkheden en toesteleigenschappen. Deze almaar toenemende heterogeniteit in de multimediaketen brengt een aantal nieuwe uitdagingen met zich mee. Een voorbeeld van een dergelijke uitdaging is de realisatie van *Universele Multimediatoegang*, wat betekent dat multimediale inhoud door eender welk toestel overal en te allen tijde moet kunnen afgespeeld worden.

De succesvolle totstandkoming van alomtegenwoordige en naadloze toegang tot digitale multimediapresentaties vergt een gepaste reactie vanuit verschillende kennisdomeinen. Dit wordt eveneens erkend door de Moving Picture Experts Group (MPEG).

- Het antwoord van de gemeenschap die in MPEG verantwoordelijk is voor de ontwikkeling van codeeralgoritmen, bestaat uit de specificatie van schaalbare of gelaagde codeerschema's. Inderdaad, de beeldsnelheid en de ruimtelijke resolutie van schaalbare videobronnen kunnen bijvoorbeeld op een eenvoudige manier aangepast worden aan de beperkingen die door een bepaalde gebruiksomgeving worden opgelegd. Hierbij kan gedacht worden aan restricties in termen van beschikbare bandbreedte en schermresolutie.

- Het antwoord van MPEG's metadatagemeenschap bestaat uit de ontwikkeling van hulpmiddelen die toelaten de eigenschappen van mediabronnen vast te leggen, alsook de beperkingen van gebruiksomgevingen. Deze beschrijvingen maken het mogelijk om een formaatonafhankelijk adaptatiesysteem te bouwen dat in staat is om de gebruikerservaring te maximaliseren, voor het afspelen van een welbepaalde multimediapresentatie in een welgedefinieerde gebruiksomgeving.

In dit doctoraatsproefschrift hebben we in eerste instantie een aantal concepten en ontwerpprincipes bestudeerd van de nieuwe H.264/MPEG-4 Advanced Video Coding-standaard, waarvan de naam gewoonlijk wordt afgekort tot H.264/AVC. Deze specificatie belichaamt de laatste ontwikkelingen in het domein van gestandaardiseerde digitale videocodering. Onze experimenten hebben aangetoond dat H.264/AVC, in vergelijking met vorige compressiestandaarden, tot 50% beter doet in termen van codeerefficiëntie. Deze vaststelling is in lijn met de resultaten die bekomen werden in andere experimenten, zoals besproken in de wetenschappelijke en technische literatuur.

Het ontwerp van de H.264/AVC-standaard wordt, naast efficiëntie, ook gekenmerkt door een grote flexibiliteit. Dit laat het gebruik toe van H.264/AVC-gecodeerde bitstromen in verschillende types netwerken en applicatiedomeinen[2]. In deze context werden een aantal hulpmiddelen onderzocht die aanwezig zijn in de eerste versie van de H.264/AVC-specificatie. Deze hulpmiddelen kunnen aangewend worden om videostromen aan te passen aan de beperkingen van verschillende gebruikscontexten. Meer bepaald kan hierbij gedacht worden aan meerlagige temporele schaalbaarheid en Flexibele Macroblokordening (Eng. *Flexible Macroblock Ordering*; FMO) voor de codering van interessegebieden (Eng. *Regions Of Interest*; ROIs).

De klemtoon van onze studie werd gelegd op het aanreiken van een volledig overzicht met betrekking tot het gebruik van meerlagige temporele schaalbaarheid in H.264/AVC-bitstromen. Bijgevolg bevat dit overzicht een uitgebreide uitleg over subsequenties (Eng. *sub-sequences*) en subsequentielagen (Eng. *sub-sequence layers*), hiërarchische codeerpatronen, en metadataboodschappen (Eng. *Supplemental Enhancement Information messages*; *SEI messages*) voor het communiceren van de bitstroomstructuur naar adaptatielogica of een decoder. Dankzij het gebruik van subsequenties is het mogelijk de codeervrijheid van H.264/AVC op een minimale manier te beperken zodat gecodeerde bitstromen op een zinvolle manier kunnen aangepast worden langsheen de temporele dimensie.

[2]Efficiënt en flexibel zijn twee termen die vaak gebruikt worden in de context van de H.264/AVC-standaard.

Meer uitgebreide adaptatiemogelijkheden worden op het moment van schrijven vastgelegd in een nieuw amendement op de H.264/AVC-specificatie. Naar dit amendement wordt typisch verwezen als H.264/AVC Schaalbare Videocodering (Eng. *Scalable Video Coding*; SVC). Deze uitbreiding op de H.264/AVC-standaard bevat uitdrukkelijke ondersteuning voor resolutie- en kwaliteitsschaalbaarheid; de mogelijkheden qua temporele schaalbaarheid worden overgenomen uit de eerste versie van de H.264/AVC-specificatie.

In deze thesis hebben we ook toegelicht hoe mediabronnen met behulp van bitstroomsyntaxbeschrijvingen (Eng. *Bitstream Syntax Descriptions*; BSDs) op een formaat-onafhankelijke manier kunnen aangepast worden in functie van de beperkingen van een gebruiksomgeving. Een BSD bevat een beschrijving van de hoog-niveaustructuur van een binaire mediabron, typisch vastgelegd met behulp van XML (Eng. *eXtensible Markup Language*). Eénmaal een XML-gebaseerde beschrijving van de structuur van een binaire mediabron bekomen werd (i.e., de BSD), kan deze in een volgende stap getransformeerd worden om een gewenste aanpassing van de betreffende mediabron te weerspiegelen. De getransformeerde BSD kan vervolgens gebruikt worden om automatisch een aangepaste versie van de mediabron te creëren, en dit met behulp van formaatonafhankelijke adaptatielogica. De aangepaste mediabron is dan geschikt voor consumptie in een welbepaalde gebruiksomgeving.

Twee verschillende technieken voor het BSD-gedreven adapteren van binaire mediabronnen werden in meer detail bestudeerd: enerzijds een gestandaardiseerd raamwerk gebaseerd op de *MPEG-21 Bitstream Syntax Description Language* (MPEG-21 BSDL) en anderzijds een raamwerk gebaseerd op het gebruik van de *Formal Language for Audio-Visual Object Representation*, uitgebreid met XML-ondersteuning (XFlavor). Gegeven een formele beschrijving van de syntax van een welbepaald mediaformaat (in MPEG-21 BSDL of XFlavor), beschikken beide technologieën over verschillende mogelijkheden om de structuur van een binaire mediabron automatisch te vertalen naar een XML-beschrijving, en voor het opeenvolgend aanmaken van een op maat gemaakte bitstroom met behulp van een getransformeerde BSD.

De hoog-niveaustructuur van een aantal veelgebruikte formaten voor videocodering en opslag werd beschreven met behulp van BSDL en XFlavor. De opbouw van een BSDL-beschrijving voor de eerste versie van de H.264/AVC-standaard werd in meer detail besproken. De ontwikkeling van deze beschrijving vereiste immers een aantal niet-normatieve uitbreidingen op BSDL, nodig voor het oplossen van verschillende tekortkomingen in de betreffende taal.

Naast het testen van de uitdrukkingskracht van MPEG-21 BSDL en XFlavor, hebben we eveneens hun prestaties geëvalueerd in de context van een aantal mediaformaten, ons hierbij richtend op applicaties zoals de BSD-gebaseerde uitbuiting van temporele schaalbaarheid en BSD-gedreven demultiplexering. Onze experimenten hebben geleid tot de identificatie van een aantal fundamentele prestatieproblemen, in het bijzonder de trage en geheugenbelastende creatie van BSDs door BSDL's BintoBSD Parser (die toelaat om op een formaatonafhankelijke en automatische manier BSDs voort te brengen), de omvang van de BSDs zoals aangemaakt door XFlavor-gebaseerde parsers, en de geheugenintensieve transformatie van BSDs door gebruik te maken van *eXtensible Stylesheet Language Transformations* (XSLTs).

De prestatieproblemen van BSDL's BintoBSD Parser zijn te wijten aan het bijhouden van de volledige BSD in het systeemgeheugen, wat nodig is om een correct gedrag van de betreffende parser te garanderen. Teneinde een meer efficiënte generatie van BSDs mogelijk te maken, hebben we BFlavor (BSDL + XFlavor) voorgesteld. Deze nieuwe techniek voor het aanmaken van BSDs is het resultaat van een kruisbestuiving tussen MPEG-21 BSDL en XFlavor. Inderdaad, BFlavor harmoniseert beide talen door het samenbrengen van hun sterktes en door het wegwerken van hun zwaktes. Meer bepaald waren de verwerkingsefficiëntie en de uitdrukkingskracht van XFlavor, samen met de mogelijkheid van MPEG-21 BSDL om hoog-niveau BSDs aan te maken, onze belangrijkste motieven voor de ontwikkeling van BFlavor. Bijgevolg is het gebruik van BFlavor-gebaseerde BSD-producenten, die formaatspecifiek zijn maar automatisch gegenereerd worden door een formaatonafhankelijk proces, een efficiënt alternatief voor het gebruik van BSDL's formaatneutrale maar inefficiënte BintoBSD Parser. De ontwikkeling van BFlavor kan beschouwd worden als de hoofdbijdrage van dit onderzoek.

De uitdrukkingskracht en prestaties van een hybride, BFlavor-gedreven adaptatieketting voor binaire mediabronnen werden onderzocht met behulp van verschillende experimenten. Eén reeks experimenten beoogde het uitbuiten van meerlagige temporele schaalbaarheid in H.264/AVC. Bijzondere aandacht werd besteed aan het gezamenlijk gebruik van subsequenties en SEI-boodschappen. De bekomen resultaten werden vergeleken met technieken die volledig gebaseerd zijn op ofwel BSDL of XFlavor. De hybride ketting gebaseerd op BFlavor was de enige die een elegante en praktische oplossing kon aanreiken voor de BSD-gedreven aanpassing van H.264/AVC-bitstromen in het temporeel domein.

Verder hebben we in deze dissertatie ook stilgestaan bij de BSD-gebaseerde constructie van plaatsvervangende sneden (Eng. *slices*) en beelden, en dit in de context van verschillende formaten voor digitale videocodering. Deze kunstmatige syntaxstructuren laten toe om een aantal ongewenste neveneffecten weg te werken. Deze neveneffecten zijn het resultaat van een BSD-gebaseerde adaptatiestap in het gecomprimeerde domein. Het gebruik van plaatsvervangende sneden en beelden werd in meer detail besproken in de context van de BSD-gebaseerde uitbuiting van temporele en ROI-schaalbaarheid in de eerste versie van de H.264/AVC-standaard, respectievelijk een oplossing aanreikend voor synchronisatieproblemen en het overtreden van regels opgelegd door de betreffende specificatie.

Een laatste bijdrage van dit onderzoek bestaat uit het introduceren van een ware-tijdsaanpak voor de BSD-gedreven adaptatie van H.264/AVC-bitstromen in het temporeel domein. De volgende technologieën werden gebruikt: BFlavor voor de efficiënte generatie van BSDs, Streaming Transformations for XML (STX) voor de efficiënte transformatie van BSDs, en BSDL's formaatneutrale BSDtoBin Parser voor de efficiënte constructie van op maat gemaakte videobitstromen. Uitgebreide prestatiemetingen werden voorgesteld voor verschillende gevalstudies, in het bijzonder voor het uitbuiten van temporele schaalbaarheid door het weglaten van sneden, het uitbuiten van temporele schaalbaarheid door het gebruik van plaatsvervangende sneden, en de aanmaak van samengevatte videobitstromen (Eng. *video skims*). De laatste toepassing is mogelijk dankzij het verrijken van een BSD met extra metadata die door het BSD-transformatieproces kunnen gebruikt worden.

Alhoewel beperkt in zijn toepassingsgebied, hopen we de lezer overtuigd te hebben dat dit onderzoek een originele bijdrage heeft geleverd tot het overbruggen van de kloof tussen inhoud en context, hierbij de visie ondersteunend dat uiteindelijk de gebruiker, en niet het toestel of het netwerk, moet beschouwd worden als het eigenlijke aandachtspunt in de multimediaketting.

List of abbreviations

ADTE	Adaptation Decision Taking Engine
API	Application Programming Interface
AVS	Audio and Video coding Standard
ASP	Advanced Simple Profile
AVC	Advanced Video Coding
BiM	Binary MPEG format for XML
BDA	Blu-ray Disc Association
BFlavor	BSDL + XFlavor
BS Schema	Bitstream Syntax Schema
BSD	Bitstream Syntax Description
BSDL	Bitstream Syntax Description Language
CfP	Call for Proposals
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CGS	Coarse-Grained quality Scalability
CIF	Common Intermediate Format
CSS	Cascading Style Sheet
DCT	Discrete Cosine Transform
DIA	Digital Item Adaptation
DIM	Decoder Initialization Metadata
DOM	Document Object Model
DPB	Decoded Picture Buffer
DTM	Document Table Model
DVD	Digital Versatile Disc
EBDU	Embedded Bitstream Data Unit
ESS	Extended Spatial Scalability
FCD	Final Committee Draft
FGS	Fine-Grained quality Scalability
FLAVOR	Formal Language for Audio-Visual Object Representation
FMO	Flexible Macroblock Ordering
FourCC	Four Character Code
FRExt	Fidelity Range Extensions
gBS Schema	generic Bitstream Syntax Schema
gBSD	generic Bitstream Syntax Description

GMC	Global Motion Compensation
GNU	GNU is Not Unix
GOP	Group Of Pictures
GPAC	GNU Project on Advanced Content
GPRS	General Packet Radio Service
HD	High Definition
HDTV	High Definition Television
IDR	Instantaneous Decoding Refresh
IEC	International Electrotechnical Commission
IMBR	Random Intra Macroblock Refresh
IP	Internet Protocol
I/O	Input/Output
ISO	International Organization for Standardization
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
JM	Joint Test Model of the JVT
JND	Just Noticeable Difference
JPEG	Joint Photographic Experts Group
JSVM	Joint Scalable Video Model
JTC	Joint Technical Committee
JVT	Joint Video Team of VCEG and MPEG
Kbps	Kilobits per second
Mbps	Megabits per second
MC-EZBC	Motion Compensated Embedded Zero Block Coding
MCP	Motion-Compensated Prediction
MDS	Multimedia Description Schemes
MMCO	Memory Management Control Operation
MOS	Mean Opinion Score
MPEG	Moving Picture Experts Group
MSE	Mean Squared Error
MTU	Maximum Transfer Unit
N-VOP	non-coded VOP
NAL	Network Abstraction Layer
NALU	NAL Unit
PC	Personal Computer
PDA	Personal Digital Assistant
POC	Picture Order Count
PPS	Picture Parameter Set
PRExt	Professional Range Extensions
PSNR	Peak Signal-to-Noise Ratio
PU	Parse Unit
QCIF	Quarter CIF
QoE	Quality of Experience
QoS	Quality of Service
QP	Quantization Parameter

RBSP	Raw Byte Sequence Payload
RD	Rate-Distortion
ROI	Region Of Interest
RTP	Real-time Transport Protocol
SAX	Simple API for XML
SC	Steering Committee
SDTV	Standard Definition Television
SEI	Supplemental Enhancement Information
SMPTE	Society of Motion Picture and Television Engineers
SNR	Signal-to-Noise Ratio
SPS	Sequence Parameter Set
STB	Set-Top Box
STX	Streaming Transformations for XML
SVC	Scalable Video Coding
UED	Usage Environment Description
UEP	Unequal Error Protection
UMA	Universal Multimedia Access
VC-1	Video Codec 1
VCEG	Video Coding Experts Group
VCL	Video Coding Layer
VLC	Variable Length Code
VOP	Video Object Pane
VTC	Visual Texture Coding
VUI	Video Usability Information
W3C	World Wide Web Consortium
WiFi	Wireless Fidelity
XFlavor	Flavor, extended with XML features
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	Extensible Stylesheet Language Transformations

Contents

Chapter 1

Introduction

Variety is the very spice of life that gives it all its flavour.

William Cowper (1731-1800), The Task, 1785.

1.1 Context

The last decade has witnessed a significant increase in the use of digital multimedia content [1]. This growth has led to a vast diversity of media formats, enabling an increased richness of audio-visual content and services. Several formats for digital video coding are for instance the driving force behind user-centric content sites such as YouTube[1] and Metacafe[2], which allow users to upload, view, and share video feeds with a minimal effort.

At the same time, the multitude of devices for accessing and interacting with multimedia content has grown substantially. This spectrum ranges from Personal Computers (PCs) and Set-Top Boxes (STBs) to portable devices like Personal Digital Assistants (PDAs) and mobile phones. Each device may be constrained in a different way, such as in terms of display size, color support, input options, processing power, memory resources, and power supply.

A wide spectrum of networks for the transmission of multimedia content has emerged as well. In particular, wireless networks and broadband access technologies have proliferated in recent years, giving users the possibility to access the Web and multimedia content from different locations, in different contexts, and with varying connectivity characteristics.

[1]Available online: `http://www.youtube.com`.

[2]Available online: `http://www.metacafe.com`.

1.1.1 Scalable video coding

In recent years, scalable coding has been one of the key research topics in the field of digital media coding. Scalable bitstreams can be used for various purposes, such as the adjustment of the transmitted bit rate according to the prevailing network throughput in streaming applications, or for scaling the complexity of the decoding process according to the available computational resources on a terminal. As such, scalable content allows to take into account the capabilities and the constraints of different usage contexts.

Scalable coding partitions a coded bitstream into a number of layers with a different impact on the decoded quality. Typically, the lowest layer is coded independently, while each subsequent layer is coded with respect to previously decoded layers. The lowest layer is called the base layer; the other layers are denoted as enhancement layers. Therefore, scalable coding is sometimes referred to as layered or hierarchical coding. As a rule of thumb, the more layers that are taken into account by the decoding process, the better the user experience.

Scalable video coding techniques are usually classified into temporal, spatial, and Signal-to-Noise Ratio (SNR) approaches, as well as any combination of these (known as combined or hybrid scalability). The different scalability axes are visualized in Figure 1.1. Each axis has a different impact on the perceived quality:

- **temporal scalability:** the more layers decoded, the higher the temporal resolution;
- **spatial scalability:** the more layers decoded, the higher the spatial resolution;
- **SNR or fidelity scalability:** the more layers decoded, the better the visual quality (i.e., the less visual artifacts).

Using this layered approach, the quality of scalable bitstreams can typically be reduced by means of straightforward editing operations. The resulting bitstreams are characterized by a lower quality, but also by the fact that their decoding requires less network and terminal resources.

Scalable coding tools are available in several standards for digital video coding, such as H.262/MPEG-2 Video [2], H.263 [3], and MPEG-4 Visual [4]. However, thus far, the commercial use of scalable video coding has not been successful. A number of possible reasons can be identified:

- licensing issues;

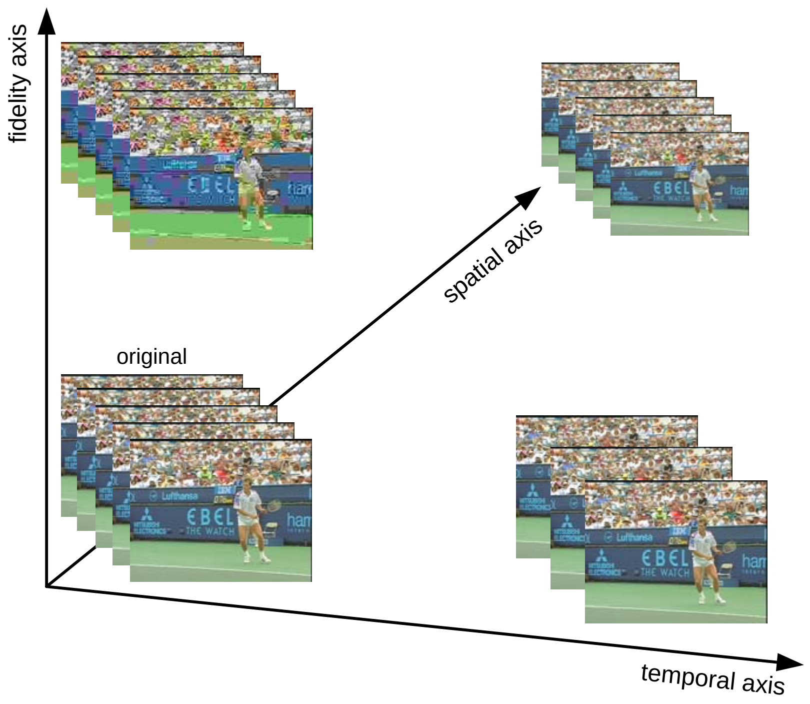


Figure 1.1: The different scalability axes in digital video coding (using the Stefan test sequence).

- the limited application space;
- the lack of coding efficiency;
- the complexity of the coding algorithms;
- the lack of an overall framework that facilitates the straightforward adaptation of scalable media content.

Nowadays, the need for scalable content has soared significantly, thanks to the fast evolution of the terminal and network technology during the last years, resulting in a plethora of devices and communication possibilities.

Furthermore, in this dissertation, a state-of-the-art video coding format is studied in more detail. This specification is at the foundation of a newly developed standard for scalable video coding, which targets a coding-efficient solution with an acceptable complexity (1.5 times the decoding complexity of single layer H.264/AVC decoding at a particular resolution and bit rate) [5].

In this research, attention is also paid to an efficient, secure, and interoperable framework for the consumption of multimedia content in heterogeneous usage environments. The vision and goals of this framework are briefly introduced in the next section, with a particular focus on the format-agnostic adaptation of digital media content.

1.1.2 MPEG-21 Digital Item Adaptation

The MPEG-21 Multimedia Framework aims at easily exchanging multimedia content without technical barriers, by shielding users from network and terminal installation, management, and implementation issues [6]. More precisely, the goal of this framework is to realize the Universal Multimedia Access (UMA; [7]) paradigm, which is the ability to achieve seamless access to multimedia content at any place, at any time, and with any device.

The MPEG-21 Multimedia Framework is organized into several independent parts. One pillar comprises the MPEG-21 Digital Item Adaptation (MPEG-21 DIA; [8]) standard, which specifies a comprehensive set of description tools that enable the optimized adaptation of multimedia content. DIA descriptions are specified in the widely used eXtensible Markup Language (XML; [9]), facilitating the development of applications that are creating, aggregating, exchanging, and consuming these descriptions.

Descriptions of the usage environment are employed for determining what version of the (scalable) content optimally fits this context. For this purpose, DIA provides a set of metadata describing the context in terms of network and device characteristics, user preferences, and natural environment characteristics (e.g., location information). These metadata are gathered under a cluster called Usage Environment Descriptions (UEDs).

Additionally, DIA specifies content-centric metadata facilitating the content adaptation process itself.

1. DIA provides tools to express the correspondence between the usage context and the characteristics of the content. These descriptions assist in the decision-making process regarding what adaptation operations to perform. That way, it is possible to obtain a media resource that is optimally tailored for a particular usage environment. For instance, the *Terminal and Network Quality of Service* (QoS) cluster within DIA specifies tools to describe the relationship between QoS constraints (e.g., on the network bandwidth or a terminal's computational capabilities), feasible adaptation operations satisfying these constraints, and associated media resource qualities that result from a possible adaptation step.

2. DIA provides the means to perform content adaptation in a format-independent way. In particular, MPEG-21 DIA leverages the use of scalable media resources by specifying generic tools for describing their high level syntax in XML. Such an XML description can subsequently be used by a format-agnostic content adaptation engine to obtain an adapted version of the scalable media resource.

One of DIA's bitstream syntax description tools will be extensively studied in this dissertation, as it bridges the gap between the flexibility offered by (scalable) media resources on the one hand and the constraints of a particular usage context on the other hand [10].

1.2 Outline

The outline of this dissertation is as follows. In the next chapter, the H.264/AVC standard is discussed in more detail. This specification for digital video coding is designed to provide an efficient and flexible solution that is appropriate for use in a broad range of applications, ranging from video coded at postage stamp resolution to high-definition content suited for playback in a digital cinema environment. The third chapter subsequently examines a number of adaptivity provisions that are incorporated in the first version of the H.264/AVC standard. The emphasis is put on a discussion regarding the different techniques that can be used for the exploitation of temporal scalability.

In chapter four, we study how XML-based bitstream syntax descriptions can be employed as an intermediate layer for the adaptation of (scalable) bitstreams to the constraints imposed by a particular usage environment. Two XML-based tool chains are discussed in more detail, both enabling the description-driven adaptation of binary media resources. An analysis is provided of their expressive power and computational complexity, targeting the temporal adaptation of bitstreams compliant with the VC-1 video coding standard. Our study resulted in the identification of a number of issues. Solutions for these problems are addressed in the next chapters.

Chapter five introduces a new tool for the efficient generation of XML-based bitstream syntax descriptions. This description tool harmonizes the two solutions that were explained in the previous chapter by combining their advantages and by eliminating their disadvantages. Its efficiency is tested by the exploitation of temporal scalability in the context of the MPEG-1 Video and H.264/AVC coding formats. These experiments show that our new description tool offers an adequate solution for a fundamental performance problem identified in the previous chapter.

The sixth chapter describes an enhanced approach towards the description-driven adaptation of coded video bitstreams. This technique, which is based on the use of placeholder pictures, allows the elimination of a number of unwanted side effects. These side effects stem from the fact that content adaptation is executed in the compressed domain. The use of our enhanced approach is verified for the description-driven adaptation of H.264/AVC bitstreams in the temporal and spatial dimension. Finally, we end this dissertation with the major conclusions that can be drawn from our research.

The research that has led to this dissertation resulted in a number of scientific and technical publications. One paper has been accepted for publication in Elsevier's *Signal Processing: Image Communication* [11]. Two papers were published in the *Lecture Notes in Computer Science* series [12, 13], which appears in the Science Citation Index (SCI). Furthermore, our work also served as a contribution to a paper that was published in *IEEE Transactions on Circuits and Systems for Video Technology* [14], to a paper that was published in *Journal of Visual Communication & Image Representation* [15], to a paper that was published in *Multimedia Systems Journal* [16], to four papers that were published in *Lecture Notes in Computer Science* [17–20], and to a paper that is accepted for publication in *Journal of Visual Communication & Image Representation* [21]. In addition, seven contributions were made as a first author to papers presented at international conferences [22–28], as well as twelve contributions as a co-author [29–40]. Finally, our research has also resulted in a number of MPEG input contributions [41–49].

Chapter 2

The H.264/AVC standard

Your codec is a guest in my process space – I expect it to act accordingly, not to terminate my app.

Avery Lee, author of VirtualDub, regarding anti-debugger routines added to the DivX 5.1 video codec.

2.1 Introduction

Digital video coding is used under the hood of a plethora of multimedia applications, including digital storage media, streaming video, television broadcasting, and many others. The engines behind the commercial success of these applications are several international specifications for video coding. The key merit of these standards is worldwide interoperability among products developed by different manufacturers, while at the same time allowing enough flexibility for tailoring the coding technology to fit the needs of a certain multimedia application.

H.120 was the first international standard for digital video coding to see the light in 1984 [50]. Since then, numerous coding efficiency improvements have been developed by the academic world and the industry, especially by taking advantage of an ever increasing processing power. This increase in processing power allowed the use of more complex algorithms, which were mainly used to further minimize the large amount of temporal redundancy in video content.

Some of these innovations have found their way into new and well-defined sets of coding tools through an open process of collaboration. This resulted in the standardization of H.261 in 1990 [51], MPEG-1 Video in 1993 [52], H.262/MPEG-2 Video in 1994, H.263 in 1995, and MPEG-4 (Part 2) Visual in 1999. The main incentive behind the development of these specifications was the need for a higher coding efficiency, which has always been a trade-off between bit rate, distortion, and computational complexity. For a more detailed overview regarding the history and the concepts of digital video coding, we would like to refer the reader to a manuscript of Sullivan *et al.* [53]. A good introduction to the principles of video compression can also be found in [54].

H.264/AVC is currently the most powerful and state-of-the-art specification in the series of international video coding standards [55]. The main objectives behind its development are as follows [56]:

- an enhanced coding efficiency;
- an improved use of coded video data in a wide variety of network environments (focusing on mobile networks and the Internet);
- a simple syntax specification (targeting simple and clean solutions).

In this chapter, we first discuss the history of the H.264/AVC standardization process, followed by a brief overview of the profiles defined in H.264/AVC. Next, we outline the most important design characteristics of the H.264/AVC specification. Finally, we discuss our main contribution in this domain, in particular an assessment of the rate-distortion performance of H.264/AVC compared to a number of other video coding formats.

H.264/AVC specifies an extensive quantity of features, out of which we discuss the ones that are essential for achieving a good understanding of our research. For a more detailed discussion regarding the H.264/AVC standard, we would like to refer the interested reader to a special issue of *IEEE Transactions on Circuits and Systems for Video Technology* [57] and to a special issue of *Journal of Visual Communication & Image Representation* [58].

2.2 History of the standardization process

2.2.1 The first version of H.264/AVC

The initial version of H.264/AVC was developed over a period of about four years [59]. The roots of the H.264/AVC standard lie in the H.26L project initiated in 1998 by the Video Coding Experts Group (VCEG) of the ITU Telecommunication Standardization Sector (ITU-T), one of the three sectors of the International Telecommunication Union (ITU). The H.26L project aimed at the creation of a "long-term" recommendation for digital video coding, with the target to double the coding efficiency compared to any other standard for digital video coding. This essentially comes down to halving the bit rate necessary for a given level of picture fidelity, which is usually expressed in terms of Peak Signal-to-Noise Ratio (PSNR; see Section 2.5).

In 2001, ISO/IEC's Moving Picture Experts Group (MPEG) issued a Call for Proposals (CfP) to invite new contributions to further improve the coding efficiency beyond what was achieved on the then recently finished MPEG-4 Visual project. VCEG chose to provide the draft design for its new standard in response to MPEG's CfP. Other proposals were also submitted and were tested by MPEG as well. As a result of those tests, ITU-T and ISO/IEC agreed to jointly develop a new video coding standard and to use H.26L as the starting point.

A Joint Video Team (JVT), consisting of experts from VCEG and MPEG, was formed in December 2001 and it completed the technical development of the new standard in May of 2003. The ITU-T approved the standard under its naming structure as ITU-T Recommendation H.264, and ISO/IEC approved it as ISO/IEC 14496-10 Advanced Video Coding (AVC[1]), under the MPEG-4 umbrella of standards. The name of the specification is commonly abbreviated as H.264/AVC or H.264/MPEG-4 AVC to reflect the cooperation between the two organizations.

2.2.2 The Fidelity Range Extensions

The initial H.264/AVC standard was primarily focused on consumer-quality video, based on Standard Definition Television (SDTV) or lower video resolutions (i.e., progressive video content having a resolution lower than 1280×720, 720p), eight bits per sample, and 4:2:0 chroma sampling. The specification did not include support for use in the most demanding professional applications, such as studio editing and post-processing, for which it is necessary to support alternative color sampling structures and a higher sample accuracy [60].

To address the needs of these most-demanding applications, the JVT added a number of extensions to the capabilities of the original standard (see Section 2.3). This standardization effort started in May of 2003 and was completed in August of 2004. The extensions are incorporated in a first amendment to the H.264/AVC specification and are better known as the Fidelity Range Extensions (FRExt, Amendment 1 of H.264/AVC)[2].

[1]The abbreviation AVC is not to be confused with the Apple Video Compressor codec (short for compression/decompression), which is also denoted as AVC and which is sometimes nicknamed "Road Pizza". The Apple Video Compressor was developed in the early nineties by Apple Computer for the QuickTime architecture.

[2]These extensions were initially known as the Professional Range Extensions (PRExt). However, the name was changed to FRExt in order to avoid giving the impression that the prior standardization effort was not carried out in a professional way.

Besides the support for other color sampling formats and an extended sample depth, FRExt also introduces an 8×8 transform block size [61]. This transform block size is adaptively selectable by an encoder in addition to the 4×4 transform block size specified in the first version of the H.264/AVC standard. Thanks to a better preservation of details such as film grain and fine textures, the adaptive block size transform is of particular interest for the encoding of high-definition content (HD content). Finally, FRExt also offers support for a number of new color spaces, such as the YCoCg color space (luma, chroma orange, chroma green). This color space is characterized by a simple set of transform equations relative to the traditional RGB color space (red, green, blue). YCoCg-R (reversible YCoCg) is a variant of the YCoCg color space that allows a lossless round trip through RGB at the expense of a higher bit depth for the chroma channels [26, 62].

The FRExt amendment has resulted in a third version of the H.264/AVC specification, as the second edition refers to a version containing a number of corrections of errata, but no added features relative to the first version. A fourth version includes more errata corrections.

2.2.3 The scalable extensions to H.264/AVC

At the time of writing this dissertation (the summer of 2006), the JVT is working on the development of a new set of extensions to H.264/AVC. This work item addresses the desire to incorporate algorithms for Scalable Video Coding (SVC) into the design of H.264/AVC in a maximally-compatible way[3]. The term "scalability" refers to a functionality that allows the removal of particular parts of a coded bitstream while still achieving a rate-distortion performance with the remaining data (at any supported spatial, temporal, or SNR resolution) that is comparable to single-layer H.264/AVC coding (at that particular resolution). The term "comparable" refers to the design goal to achieve a coding efficiency within about 10% excess bit rate for the same decoded video fidelity [5]. The intent is to finish the drafting work on the new extensions by the start of 2007. A more in-depth overview of the scalable extensions to H.264/AVC is provided in Chapter 3.

2.3 Profiles

To manage the large number of coding tools included in H.264/AVC, as well as to keep order in the broad range of formats and bit rates supported, the con-

[3]The first version of H.264/AVC is sometimes referred to as a fully scalable standard by marketing documents. However, this is due to its support for the entire bandwidth spectrum.

cept of profiles and levels is employed. A profile defines a set of coding tools or algorithms that can be used in generating a compliant bitstream, whereas a level places constraints on certain key parameters of the bitstream, such as the picture resolution and the bit and frame rate. To address the applications considered by the two standardization organizations, three profiles were established in the first version of the H.264/AVC specification.

- The **Baseline Profile (BP)** was designed to minimize complexity on the one hand, and to provide high robustness and flexibility for use over a broad range of network environments and conditions on the other hand. It aims at (low-delay) applications in the field of video conferencing and wireless communications.
- The **Main Profile (MP)** was designed with an emphasis on coding efficiency. It targets applications such as television broadcasting and digital video storage.
- The **Extended Profile (XP)** (formerly called the Streaming Profile or Profile X) is a superset of the Baseline Profile. It was designed to combine the robustness of the Baseline Profile with a higher degree of coding efficiency, a greater network robustness, and a number of enhanced modes (for instance enabling bitstream switching in video streaming applications; see Chapter 3).

The FRExt project produced a suite of four additional profiles collectively called the High or FRExt profiles: the **High Profile (HP)**, the **High 10 Profile (Hi10P)**, the **High 4:2:2 Profile (H422P)**, and the **High 4:4:4 Profile (H444P)**. These profiles support all features of the prior Main Profile and additionally support an 8×8 transform block size and perceptually-optimized encoder-selected quantization scaling matrices [61].

Figure 2.1 shows the relationship between the different profiles and the coding tools supported by the different versions of the H.264/AVC standard. A number of these coding tools will be described in more detail in the remainder of this dissertation. The High Profile is often considered to be the most important profile of the H.264/AVC standard, mainly because of its fast embrace by the industry and standardization organizations. It supports eight bit video with 4:2:0 sampling, addressing high-end consumer use and other applications using high-resolution video without a need for extended chroma formats or extended sample accuracy. The High 4:4:4 Profile, offering initial standardized support for mathematical lossless inter coding, has recently been removed from the H.264/AVC specification due to a lack of coding efficiency.

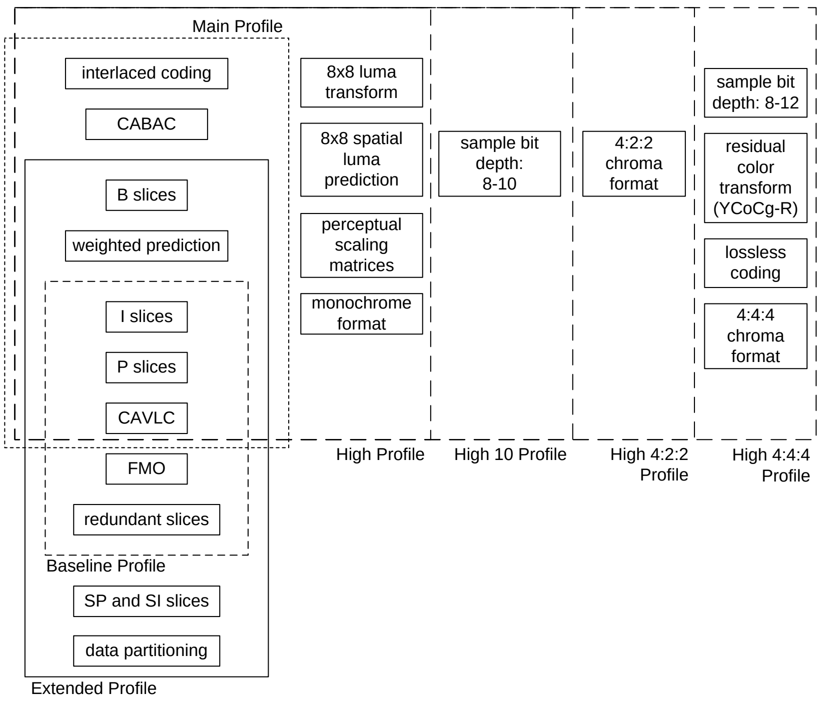


Figure 2.1: Profiles in H.264/AVC (adopted from [61] and [63]).

2.4 Technical design features

The H.264/AVC specification has a very broad application range that covers all forms of digitally coded video, from low bit-rate streaming applications to HDTV broadcast and digital cinema applications characterized by nearly lossless coding. As shown by Figure 2.2, H.264/AVC relies on a two-tier design to address the requirements of these various applications: it covers a Video Coding Layer (VCL), which is designed to efficiently represent the video content, and a Network Abstraction Layer (NAL), which formats the VCL representation of the video data and provides header information to package the coded video data for transport or storage. As such, this design allows to distinguish between coding-specific features, which is the focus of the VCL, and transport/storage-specific features, which is the focus of the NAL.

Most features outlined in the following sections are already covered by a number of high-quality publications in the scientific literature [53, 56, 64]. However, some of them will be discussed in more detail to make this dissertation more self-containing, and hence, to increase its readability.

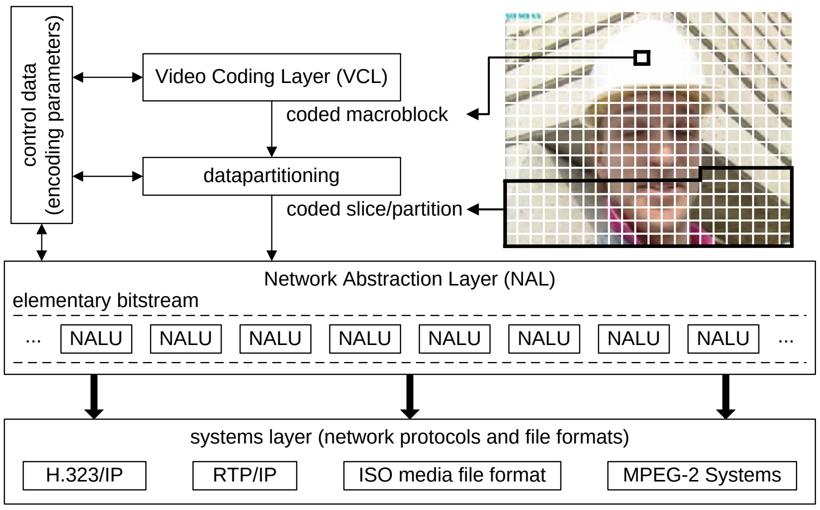


Figure 2.2: Two-tier design of H.264/AVC [56]. The screenshot is taken from the Foreman test sequence.

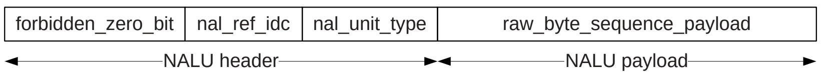


Figure 2.3: Structure of a NAL unit, which is the fundamental unit of processing of the NAL, either carrying coded video or coded header data.

2.4.1 The network abstraction layer

The NAL is designed to enable simple and effective customization of the output of the VCL for a broad variety of systems. The full degree of customization of the video content to fit the needs of each particular application is outside the scope of the H.264/AVC standard itself, but the design of the NAL anticipates a variety of such mappings. The fundamental unit of processing in the NAL is a NAL unit (NALU). Other key concepts of the NAL design are parameter sets, access units, coded video sequences, and supplemental enhancement information. A concise discussion of these building blocks is given below.

NAL units

The NAL encoder encapsulates the output of the VCL encoder into NAL units, which are suitable for transmission over packet networks [e.g., using the Real-time Transport Protocol (RTP)] or for use in packet oriented multiplex environments (e.g., MPEG-2 Transport Streams). Consequently, a NAL unit can be considered the fundamental unit of processing by an H.264/AVC encoder and decoder. As shown by Figure 2.3, a NAL unit consists of a one-byte header and a payload byte string.

- The header indicates the (potential) presence of bit errors or syntax violations in the NAL unit payload (signaled by the `forbidden_zero_bit` syntax element), information regarding the relative importance of the NAL unit for the decoding process (communicated by the `nal_ref_idc` syntax element), and the type of the NAL unit (conveyed by the `nal_unit_type` syntax element).

- The payload byte string of every NAL unit is called the Raw Byte Sequence Payload (RBSP), which is a set of data corresponding to coded video data or header information.

The different types of NAL units are usually organized in two different classes: one class contains the so-called VCL NAL units, while the other class consists of the non-VCL NAL units.

- The VCL NAL units contain the data representing the values of the samples in the pictures.

- The non-VCL NAL units contain information such as parameter sets (important header data that can apply to a large number of VCL NAL units) and Supplemental Enhancement Information messages (SEI messages). SEI messages convey timing information and other supplemental data that may enhance the usability of the decoded video signal but that are not necessary for decoding the values of the samples in the pictures.

The Annex B byte stream format Some systems (e.g., H.320, H.222.0/MPEG-2 Systems) require delivery of the entire or partial stream of NAL units as an ordered stream of bytes. For use in such systems, Annex B of the H.264/AVC standard specifies a byte stream format. In this format, each NAL unit may be prefixed by one or more zero-valued bytes, followed by a mandatory pattern of three bytes 0x000001, called a start code prefix. Such a NAL unit is called a byte stream NAL unit. The hierarchical syntax structure of an H.264/AVC elementary bitstream, having the byte stream format, is visualized in Figure 2.4. This syntax hierarchy was extensively used in the context of this research.

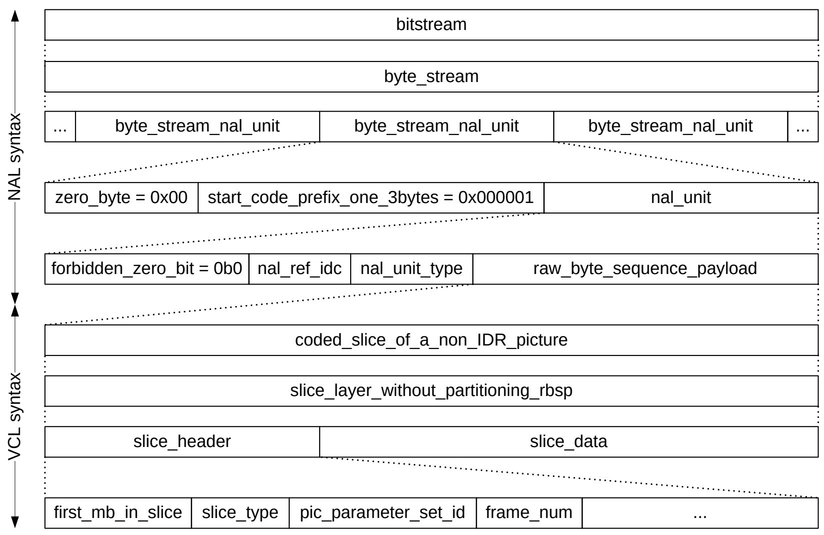


Figure 2.4: Complete syntax hierarchy for an H.264/AVC elementary bitstream having the byte stream format. The VCL syntax will be explained in more detail in Section 2.4.2.

The start code prefix can be uniquely identified in the byte stream. As shown in Figure 2.5, if necessary, the payload data of a NAL unit are interleaved with emulation prevention bytes that eliminate the occurrence of ‘pseudo’ start code prefixes within the payload, which might occur as a disadvantageous combination of coded data bytes.

The NAL unit stream format In other systems (e.g., RTP), the coded data are carried in packets that are framed by the system transport protocol, and identification of the boundaries of NAL units within the transport packets can be established without use of start code prefix patterns. As such, the transmission of start codes is typically omitted in these systems.

Access units

A set of NAL units that comprises all data necessary to decode one picture is called an Access Unit (the data of a coded picture may be distributed over different NAL units; see Section 2.4.2). This Access Unit consists of VCL NAL units composing the so-called primary coded picture. Additionally, NAL units with Supplemental Enhancement Information (SEI) can be contained in an access unit, as well as VCL NAL units for additional representation of important

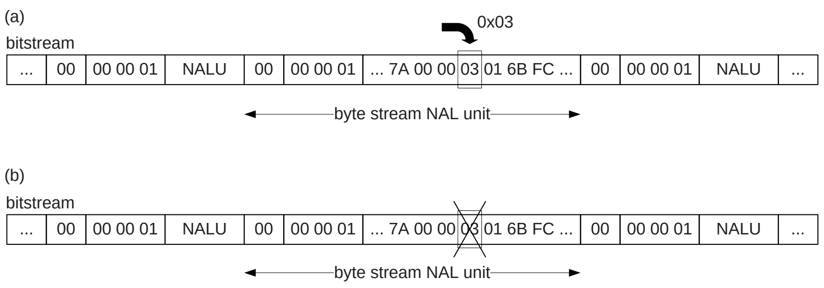

(a)
bitstream
0x03
... 00 00 00 01 NALU 00 00 00 01 ... 7A 00 00 03 01 6B FC ... 00 00 00 01 NALU ...
byte stream NAL unit
(b)
bitstream
... 00 00 00 01 NALU 00 00 00 01 ... 7A 00 00 03 01 6B FC ... 00 00 00 01 NALU ...
byte stream NAL unit

Figure 2.5: Handling of start code prefix emulation prevention bytes: (a) insertion of emulation prevention bytes by an encoder; (b) removal of emulation prevention bytes by a decoder.

areas of the primary picture for the purpose of error resilience. The latter VCL NAL units constitute so-called redundant slices.

Coded video sequences and IDR

A series of sequential access units in the NAL unit stream that requires one single Sequence Parameter Set (SPS) is called a coded video sequence. An SPS can be seen as a form of header data (see the next paragraph for a more precise definition). A NAL unit stream can contain multiple coded video sequences. Each coded video sequence begins with an access unit containing an Instantaneous Decoding Refresh (IDR).

The IDR consists of a self-contained intra-coded picture that is required to start a new video sequence. An H.264/AVC encoder cannot use any pictures that precede the IDR picture (in decoding order) as references for the inter prediction of any pictures that follow the IDR picture (in decoding order). As such, the presence of an IDR causes a reset of the different lists of reference pictures in the decoded picture buffer of a decoder.

IDR pictures provide random access points in an H.264/AVC bitstream. Furthermore, the presence of an IDR picture in an H.264/AVC bitstream also causes a reset of the Picture Order Count (POC) and the `frame_num` counters of the decoding process, while an intra-coded picture that is not an IDR picture does not. The basic concept of POC is to provide a counter that specifies the relative order of the pictures in the bitstream in output order (which may differ from the relative order in which the coded pictures appear in the data of the bitstream, which is referred to as the decoding order). The `frame_num` counter keeps track of reference pictures for error resilience purposes (see Chapter 3).

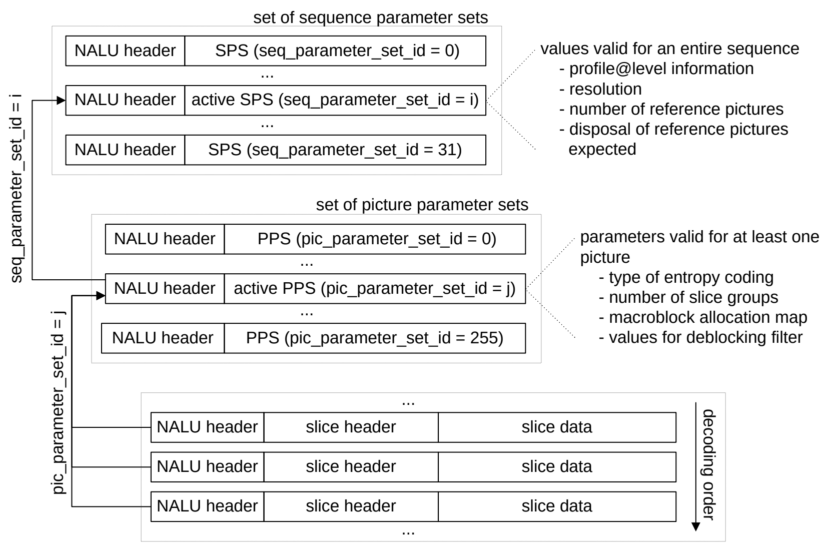

set of sequence parameter sets
NALU header
SPS (seq_parameter_set_id = 0)
...
NALU header
active SPS (seq_parameter_set_id = i)
...
NALU header
SPS (seq_parameter_set_id = 31)
values valid for an entire sequence
- profile@level information
- resolution
- number of reference pictures
- disposal of reference pictures expected
seq_parameter_set_id = i
set of picture parameter sets
NALU header
PPS (pic_parameter_set_id = 0)
...
NALU header
active PPS (pic_parameter_set_id = j)
...
NALU header
PPS (pic_parameter_set_id = 255)
parameters valid for at least one picture
- type of entropy coding
- number of slice groups
- macroblock allocation map
- values for deblocking filter
pic_parameter_set_id = j
NALU header
slice header
slice data
decoding order

Figure 2.6: Interaction between VCL NAL units, which contain coded slice data (see Section 2.4.2), and the different types of parameter sets.

Parameter sets

A parameter set contains important header information that can apply to a large number of VCL NAL units. There are two types of parameter sets:

- a Sequence Parameter Set (SPS), which applies to a series of consecutive coded video pictures; and
- a Picture Parameter Set (PPS), which applies to the decoding of one or more individual pictures.

The different parameter sets are conveyed in separate NAL units besides the VCL NAL units. There are no limits on the number of SPSs and PPSs allowed in a bitstream. In order to be able to change picture parameters such as the picture size without the need to transmit parameter set updates synchronously to the slice packet stream, the encoder and decoder can maintain a list of more than one SPS and PPS. More precisely, an H.264/AVC decoder must be capable of storing up to 32 SPSs and 256 PPSs at the same time. Note that the quantity of memory reserved for SPS and PPS storage will ordinarily be fairly small compared to the total amount of memory needed by a decoder (for example, for the storage of decoded reference pictures).

Each VCL NAL unit has an identifier that indicates the PPS to be used to decode the VCL data (i.e., the active PPS). The PPS in its turn contains an identifier to the applicable SPS (i.e., the active SPS)[4]. An active SPS remains unchanged throughout a coded video sequence (i.e., until the next occurrence of an IDR picture), and an active PPS remains unchanged within a coded picture. In this manner, a small amount of data (the identifier) can be used to establish a larger amount of information (the parameter set) without repeating that information within each VCL NAL unit. Note that it is for instance legitimate for an encoder to use only one SPS in a bitstream, and to update the SPS content associated with that identifier by repeatedly overriding the SPS with another SPS having the same identifier (instead of using multiple SPSs with different identifiers). The interaction between VCL NAL units and the different types of parameter sets is visualized in Figure 2.6.

Both types of parameter sets can be transmitted at any time, e.g. well in advance before the VCL units they apply to. Since the parameter information is crucial for the decoding of large portions of the NAL unit stream, it can be protected stronger or retransmitted at any time for increased error robustness. In-band transmission with the VCL NAL units or out-of-band transmission over a separate channel can be used if desired by a target application.

Supplemental Enhancement Information

In addition to the basic coding tools, the H.264/AVC standard enables sending extra information along with the compressed video data, called Supplemental Enhancement Information messages (SEI messages). These messages are stored in an H.264/AVC elementary bitstream using an own NAL unit type. SEI messages, introduced for the first time in H.263+, can assist in processes related to decoding, display, or other purposes. However, they are not required for the actual reconstruction of the luma and chroma samples by the decoding process. Nonetheless, their use can be made normative by an applications organization (e.g., the Blu-ray Disc Association; BDA).

SEI metadata messages can for instance be used to convey timing information, scene transition information, and arbitrary user data. These messages can also be employed to signal the occurrence of temporal enhancement layers in an H.264/AVC bitstream (see Chapter 3), or they can be used to communicate additional random access points next to IDR pictures.

[4]Some decoders activate an SPS or PPS once they retrieve these parameter sets from the bitstream, which is incorrect decoder behavior.

2.4.2 The video coding layer

The VCL contains the signal processing functionality - this is, means such as transform coding, quantization logic, Motion-Compensated Prediction (MCP), and a deblocking filter. The fundamental unit of processing of the VCL is a slice (and not a picture). Although the design of the VCL of H.264/AVC basically follows the traditional hybrid concept of block-based MCP and transform coding, a number of important innovative ideas have been developed that enable a significant improvement in terms of coding efficiency. Some of the key features are given below [61]:

- enhanced MCP capabilities (e.g., the use of variable block sizes during motion estimation for a more efficient representation of complex motion, know as tree-structured motion estimation);
- multiple reference pictures and generalized B slice coded pictures [65];
- spatial intra prediction in the pixel domain (for an efficient compression of gradients);
- 4x4 block-size transform in integer precision (allows reducing ringing artifacts, especially in areas of fine detail);
- context-adaptive in-loop deblocking filter (resulting in smooth and clean pictures due to the elimination of blocking artifacts, which are mainly the result of quantization errors);
- enhanced entropy coding methods.

The coding gains of H.264/AVC do not stem from a single coding technique, but they are the result from the combined use of advanced prediction, quantization, and entropy coding schemes. Nonetheless, the improved motion estimation and motion compensation capabilities offered by the standard can be considered the most important ones. In what follows, a number of coding tools and concepts are highlighted that contribute to a better understanding of the remainder of this dissertation.

Macroblocks, slices, and slice groups

Macroblocks. The output of the VCL is a coded video sequence, which consists of a sequence of coded pictures. Every picture is partitioned into fixed size macroblocks that each contain a rectangular picture area of 16x16 samples for the luma component and the corresponding 8x8 sample regions for each of the two chroma components (when using 4:2:0 input). Macroblocks are the basic building blocks for which the decoding process is specified. All luma and chroma samples are predicted - either spatially or temporally - and the resulting prediction residual is transmitted using transform coding: each color component of the residual is subdivided into blocks, and each block is transformed using an integer-valued transform, after which the transform coefficients are quantized and entropy coded.

Slices. The macroblocks (MBs) of a picture are organized into slices, which represent regions of the given picture that can be decoded independently. Each slice is a sequence of macroblocks that is processed in raster scan order, i.e. a scan from the top-left corner of the picture to the bottom-right corner. Note that the macroblocks are not necessarily always consecutive in the raster scan of the picture, as further described for the Flexible Macroblock Ordering (FMO) coding tool.

A picture may contain one or more slices. Each slice is self-contained, in the sense that, given the active sequence and picture parameter sets, its syntax elements can be parsed from the bitstream and the values of the samples in the area of the picture that the slice represents can basically be decoded without the use of data from other slices of the picture (provided that all previously-decoded reference pictures are available at the decoder for use in MCP). However, for completely exact decoding, information from other slices may be needed in order to apply the deblocking filter across slice boundaries. As discussed by Sullivan *et al.* in [53], slices can be used for:

- error resilience, as the partitioning of the picture allows spatial concealment within the picture and as the start of each slice provides a resynchronization point at which the decoding process can be reinitialized;
- creating well-segmented payloads for packets that fit the Maximum Transfer Unit (MTU) size of a network (e.g., the MTU size is 1500 Bytes for Ethernet, and typically around 100 bytes in wireless environments);
- parallel processing, as each slice can be encoded and decoded independently of the other slices of the picture.

As shown in Figure 2.7, the bitstream representation of a coded slice consists of a slice header and slice data. Syntax elements that change frequently from picture to picture are placed in the slice header for reasons of robustness to data losses. As such, a slice header can be considered a third parameter set, next to the sequence and picture parameter sets. Finally, the slice data contain the coded representation of the macroblocks.

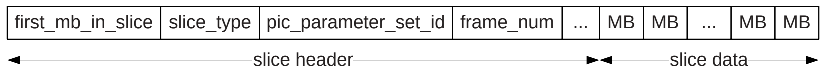


Figure 2.7: Structure of a slice, the fundamental unit of processing of the VCL.

Slice types. The H.264/AVC standard distinguishes five slice types, signaled by the value of the `slice_type` syntax element in a slice header (see Figure 2.7).

- **I slice:** A slice in which all macroblocks of the slice are coded using intra prediction.
- **P slice:** In addition to the coding types of an I slice, macroblocks of a P slice can also be coded using inter prediction with at most one MCP signal per prediction block.
- **B slice:** In addition to the coding types available in a P slice, macroblocks of a B slice can also be coded using inter prediction with two MCP signals per prediction block that are combined using a weighted average.
- **SP slice:** A so-called switching P slice is coded such that efficient and exact switching between different video streams (or efficient jumping from place to place within a single stream) is possible without the large number of bits needed for an I slice.
- **SI slice:** A so-called switching I slice allows an exact match with an SP slice for random access or error recovery purposes, while only using intra prediction.

The first three slice types listed above are very similar to coding methods used in previous standards, with the exception of the use of (multiple) reference pictures as described further in this section. The other two slice types are new. For a more profound overview regarding the novel concept of SP and SI slices, the interested reader is referred to [66]. In Chapter 3, the use of these slices will be discussed in more detail for the purpose of bit rate adaptation.

A single coded picture may consist of a mixture of the different slice types. As such, the traditional concept of I, P, and B pictures is replaced by a highly flexible and general concept that can be exploited by an encoder for different purposes. Also, from this discussion, it should be clear that a slice, and not a picture, is the fundamental unit of processing in H.264/AVC's VCL.

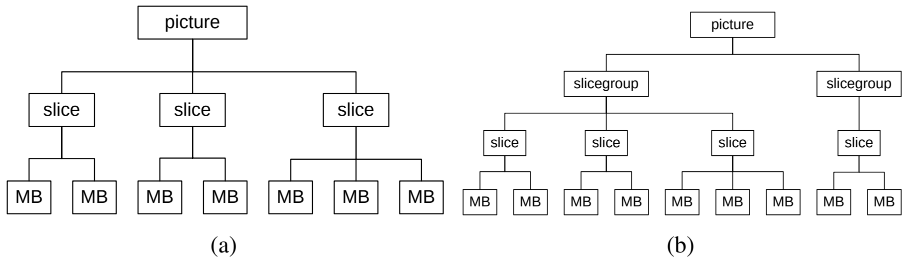
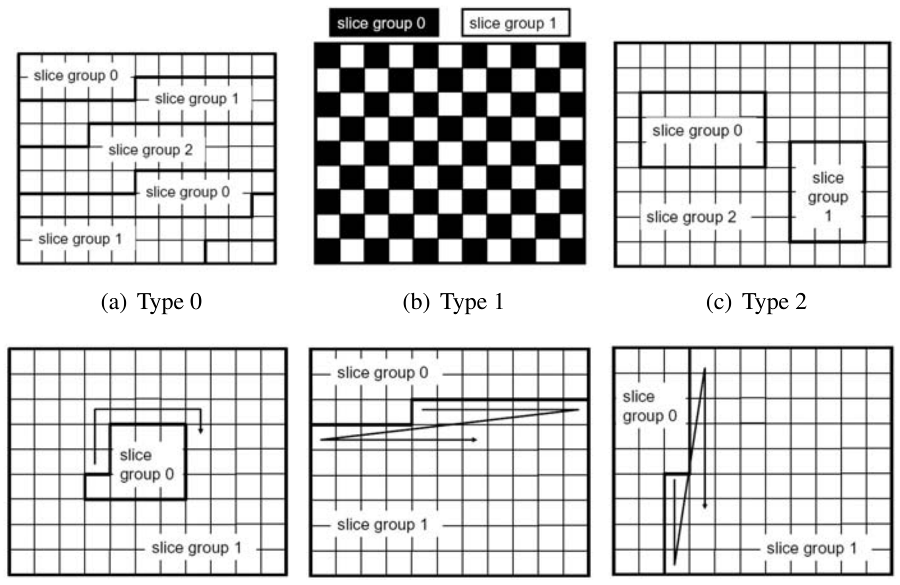


Figure 2.8: The different layers when coding a picture [68]: (a) in prior video coding standards; (b) in H.264/AVC.

Slice groups and FMO. Prior video coding standards, such as H.262/MPEG-2 Video and MPEG-4 Visual, require encoders to process the macroblocks of a picture in a consecutive raster scan order. This process typically starts at the top-left corner of a picture, encoding macroblocks row by row until the bottom-right corner of this picture is reached. In [67], Wenger and Horowitz discuss how FMO allows altering the order in which macroblocks are processed. This tool was introduced in H.264/AVC by means of a new concept called slice groups. Using FMO, pictures are no longer divided into slices but into slice groups. More precisely, each picture can be divided in up to eight different slice groups consisting of one or more slices. As illustrated by Figure 2.8, slice groups introduce a new layer between a picture and its slices.

Every macroblock can be assigned freely to a slice group using a macroblock allocation map. All macroblocks of every slice group are coded in a consecutive raster scan order. This implies that all macroblocks of a picture will be encoded in a consecutive raster scan order when assigning them to one slice group, which is behavior similar to previous standards. The term FMO is used when more than one slice group is used per picture. Since each macroblock can be assigned arbitrarily to one of the different slice groups during the encoding process, a decoder has to know which macroblock is assigned to which slice group. This is realized by transmitting a macroblock allocation map together with the coded macroblocks. This map is included inside a PPS, which is valid for a particular number of pictures.

Because up to eight slice groups can be used for a picture, up to three bits are needed for every macroblock to know to which slice group it belongs. This would make FMO an expensive feature in terms of coding efficiency. However, in most cases, certain patterns will appear in the macroblock allocation map. The regular structure of such a pattern can often be described by means of a



(a) Type 0 (b) Type 1 (c) Type 2

(d) Type 3 (e) Type 4 (f) Type 5

Figure 2.9: Different types of FMO.

simple function characterized by a number of variables. Transmitting a pattern comes down to signaling the predefined type of pattern (i.e., the function) and the variables needed to construct the macroblock allocation map. This means that the macroblock allocation map can often be stored in two to eight bytes.

The H.264/AVC specification provides seven options to store the macroblock allocation map inside a PPS syntax structure. The first six options are fixed patterns. The seventh option is used when the map cannot be represented by any of the six predefined patterns; it allows a complete randomization of the data and it should be signaled completely. Possible configurations for the six predefined patterns are shown in Figure 2.9. For this research, FMO type 2 is the most interesting configuration.

- For FMO type 0, each slice group consists of a fixed number of macroblocks which follow sequentially in raster scan order (but not necessarily consecutive). When all slice groups have been used and there are still a number of macroblocks left, the entire process is repeated starting from slice group zero.
- FMO type 1 uses a predefined function to create a scattered or dispersed pattern. The layout of the macroblock allocation map depends on the

number of slice groups. Here, the idea is to have no two macroblocks of the same slice group next to each other (for error resilience purposes).

- FMO type 2 allows defining rectangular areas on a picture. Those rectangles can be efficiently stored using the macroblock numbers of the top-left and bottom-right macroblocks. The rectangles may overlap, causing macroblocks to belong to multiple rectangles. Since this means that a macroblock would belong to multiple slice groups, which is not allowed, the macroblock is assigned to the slice group with the lowest number only. The macroblocks that do not belong to any rectangle are part of a separate slice group. Since there is a limit of eight slice groups per picture, this means that up to seven rectangles can be defined.
- FMO types 3 to 5 are known as evolving types. These types distribute the macroblocks over two separate slice groups. They are called "evolving" because the layout of the two slice groups is updated for every picture by means of a parameter conveyed in the slice header (to avoid having to send a PPS for every update step). The update process is such that the number of macroblocks increases for one slice group, while the number of macroblocks decreases in the complementary slice group.

Flexible reference picture management

Advanced motion prediction and motion compensation are two key strengths of H.264/AVC. These techniques are made possible by the adoption of several new design principles in the H.264/AVC standard [56].

- The **decoupling of picture representation methods from picture referencing capability** is a first feature that enhances the ability to predict the values of the content of a picture to be encoded. In prior standards, pictures encoded using some encoding methods (namely bi-predictively-coded pictures) could not be used as references for the prediction of other pictures in the video sequence. By removing this restriction, the new standard provides the encoder with more flexibility and, in many cases, an ability to use a picture for referencing that is a closer approximation to the picture being encoded.
- The **decoupling of the picture referencing order from the picture output order** is a second new design principle. In prior standards, there was a strict dependency between the ordering of pictures for motion compensation referencing purposes and the ordering of pictures for output (i.e., display) purposes. For instance, in H.262/MPEG-2 Video, a

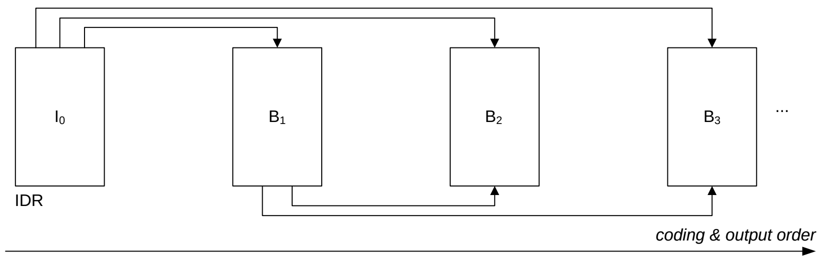


Figure 2.10: Elimination of delay when using B slice coded pictures.

B picture must be coded using a picture that precedes the B picture in output order on the one hand, and using a picture that succeeds the B picture in output order on the other hand. In H.264/AVC, these restrictions are largely removed, allowing the encoder to choose the ordering of pictures for referencing and display purposes with a high degree of flexibility, constrained only by an imposed total memory capacity bound to ensure decoding capability.

Removal of these restrictions also enables removing the extra delay previously associated with bi-predictive coding. Indeed, both the encoding and decoding delay can be reduced to a delay of zero pictures by restricting MCP from using pictures as a reference that are located in the future (in output order). This is for instance illustrated by Figure 2.10.

- A third new design feature introduced in H.264/AVC is **the use of multiple reference pictures for motion compensation**. This enables efficient coding by allowing an encoder to select on a macroblock basis - for motion compensation purposes - out of a large number of pictures that have previously been decoded and stored in the Decoded Picture Buffer (DPB).

  An additional parameter is encoded with the motion vector displacement indicating the reference picture to be used by the decoder (resulting in an additional overhead). The reference pictures may consist of I, P, and B slices. The number of applicable reference pictures is determined by the SPS. The reference picture index is not transmitted in case the application of a single reference picture is indicated.

As such, the flexible design of the H.264/AVC standard for instance allows the following coding techniques:

- coded pictures can be mixtures of I, P, and B slices;

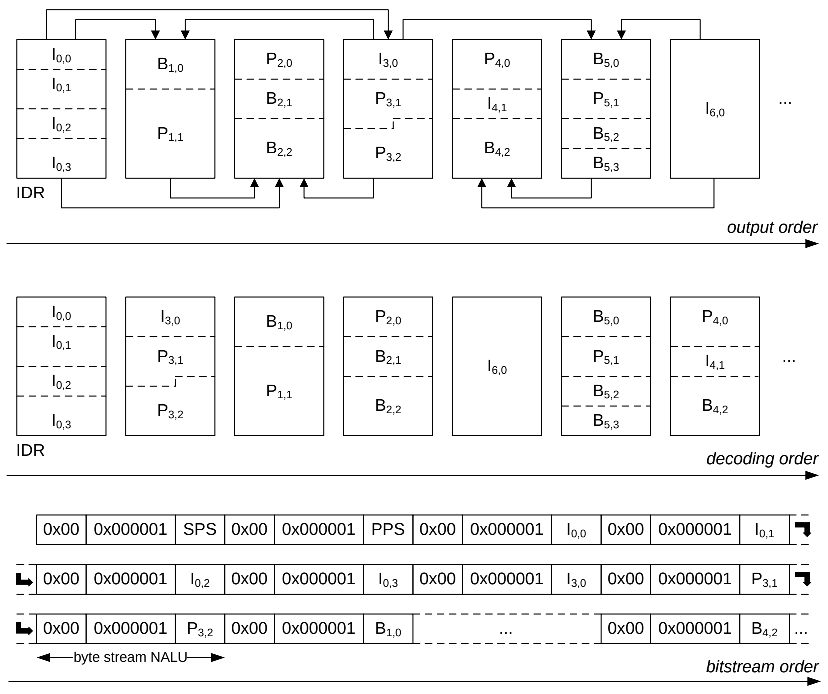


Figure 2.11: Flexible use of reference pictures.

- B slices can be used as a reference for the reconstruction of other slices;
- I and P slices can be non-reference slices;
- B slices can only have one reference picture;
- B slices can have all their references in the past (or all in the future);
- P slices can rely on many reference pictures for their reconstruction;
- delayed non-reference pictures can be stored in the DPB.

A number of the above mentioned techniques are also visualized in Figure 2.11. A label $S_{i,j}$ has the following meaning: S denotes a slice (an I, P, or B slice), while i indicates the picture number in output order and j represents the top-down order of the slice within a picture. An arrow pointing from picture A to picture B implies that the reconstruction of B is dependent on A. Finally, while the flexibility in terms of reference picture management is beneficial to the coding efficiency, it will have a serious impact on the implementation of an adaptivity feature such as temporal scalability (see Chapter 3).

Entropy coding

H.264/AVC supports two alternatives for entropy coding. These are called Context-Adaptive Variable Length Coding (CAVLC) and Context-Adaptive Binary Arithmetic Coding (CABAC). CABAC is characterized by a higher complexity than CAVLC, but it offers a better coding efficiency. Compared to CAVLC, CABAC typically reduces the bit rate by 10 to 15% for the same quality. Both entropy coding schemes operate at the macroblock level, where they are for instance employed for the representation of transform coefficients.

Table 2.1: `ue(v)`.

bit string	decoded value
0	0
010	1
011	2
00100	3
00101	4
00110	5
00111	6
...	...

Table 2.2: `se(v)`.

bit string	decoded value
0	0
010	1
011	-1
00100	2
00101	-2
00110	3
00111	-3
...	...

In both of these modes, many syntax elements are also coded using a single infinite-extent codeword set referred to as an Exponential Golomb code (Exp-Golomb code) [69]. Thus, instead of designing a different Variable-Length Coding (VLC) table for each such syntax element, only the mapping to the single codeword table is customized to the data statistics.

An Exp-Golomb code has a simple and regular structure. The definition of Unsigned and Signed Exp-Golomb codes is illustrated by Table 2.1 and Table 2.2 respectively. In the H.264/AVC standard, the Unsigned Exp-Golomb code is denoted by the descriptor `ue(v)`, while the Signed Exp-Golomb code is referred to with the descriptor `se(v)` (the `v` in the descriptor refers to the variable-length nature of the code).

Several syntax elements in the parameter sets and the slice headers are represented using Signed and Unsigned Exp-Golomb codes, which are only efficient in use when small values are expected to be more common than large values. For instance, the employment of the Unsigned Exp-Golomb code is optimal when the first input symbol, i.e. zero, has a probability of 0.5[5].

[5]For input values higher than 14, the use of an unsigned byte representation is already more efficient than the use of an Unsigned Exp-Golomb code.

2.5 Rate-distortion performance

Most video coding standards only provide the bitstream syntax and the decoding process in order to enable interoperability. The encoding process is left out of scope to permit a sufficient level of flexibility and innovation in particular implementations of the standardized video coding schemes. However, the operational control of the source encoder is a key problem in video compression. For the encoding of a video source, many coding parameters such as macroblock modes, motion vectors, and transform coefficient levels have to be determined. These values determine the rate-distortion efficiency of the bitstream produced by a given encoder.

The trade-off between coded bit rate and image distortion is an example of the general Rate-Distortion (RD) problem in communications engineering [54]. In a lossy communication system, the challenge is to achieve a target data rate with minimal distortion of the transmitted signal (in this case, a sequence of pictures). As such, the RD performance of a video codec (encoder/decoder) provides a measure of the decoded image quality (or distortion) produced at a range of coded bit rates. In the following two sections, the results of a series of experiments are reported that illustrate the RD performance gains that can be achieved when using H.264/AVC. For a good overview regarding the principles of rate-constrained coder control in the context of several video coding formats, emphasizing Lagrangian optimization techniques, we would like to refer the interested reader to [70].

2.5.1 RD performance of the first version of H.264/AVC

In this section, we discuss a number of key results regarding a series of experiments that were conducted in the course of this research. Besides gaining an insight into different coding tools, a major goal of our tests was to assess to what extent a particular requirement of H.264/AVC has been met, namely "having a capability goal of 50% or greater bit rate savings from H.263v2 (with Annexes DFIJ&T) or MPEG-4 Visual Advanced Simple Profile at all bit rates" [71]. In other words, the H.264/AVC standard should enable using half of the bits to code a video sequence at the same visual quality compared to the MPEG-4 Visual Advanced Simple Profile (ASP) specification. The assessment of this statement was done by comparing the RD performance of two implementations of the respective standards: Ad Hoc Model 2.0 (AHM 2.0) [72] as a preliminary implementation of the H.264/AVC specification and DivX 5.1[6] as a widely-used implementation of MPEG-4 Visual ASP.

[6]DivXNetworks, Inc. Available online: `http://www.divx.com/divx/`.

As input, six progressive and well-known video sequences were used in raw YCbCr 4:2:0 format. Two different resolutions were used: the Quarter Common Intermediate Format (QCIF, 176$\times$144) and the Common Intermediate Format (CIF, 352$\times$288), thus resulting in a total of twelve input video sequences. These sequences were encoded by making use of one-pass Constant Bit Rate (CBR) coding. As such, rate control is considered as a fully fledged part of an encoder (as it will often be used in practice). Rate control based on QP selection was not possible for the DivX 5.1 video codec.

Thirty different target bit rates were employed: both very low and very high bit rates to test the coding behavior in difficult circumstances. The bit rates are: 20, 40, 60, 80, 100, 200, 300, ..., 2500, and 2600 kbit/s. At each bit rate, encoding was performed at 30 frames per second. The coding pattern used is IBBBP... and an intra-coded frame is forced every 16 frames. The H.264/AVC bitstreams are conform to the Main Profile. During the generation of the DivX 5.1 bitstreams, the quarter-pel and Global Motion Compensation (GMC) features were enabled. A YUV-only graphics pipeline was maintained during the entire encoding and decoding process to avoid an impact of lossy color space conversions on the resulting quality[7].

For a given coded bit rate, the distortion of the decoded sequence is measured by making use of luma PSNR (relative to the original sequence). This measure of fidelity is the most widely-used objective video quality metric: it can be calculated quickly and measurements can be easily compared with other published results. Luma PSNR is given by the following equation:

$$\text{PSNR-Y}_{dB} = 10 \cdot \log_{10} \frac{(2^n - 1)^2}{MSE}, \tag{2.1}$$

where n is the number of bits used to represent the luma component of each pel and MSE is the Mean Squared Error. The MSE is computed as follows:

$$\text{MSE} = \frac{1}{M \cdot N} \sum_{i=1}^{M} \sum_{j=1}^{N} \left[f(i,j) - f'(i,j)\right]^2, \tag{2.2}$$

where M·N is the luma resolution of the picture and $f(i,j)$ (resp. $f'(i,j)$) is the value of the original (respectively distorted) pel.

The PSNR-Y measurements allow the registration of all changes introduced by a particular source coding algorithm in the luma domain, even changes that are not noticeable for the human visual system. Consequently, PSNR-Y can be considered a quality metric that is sometimes too objective.

[7]YUV is used in this dissertation as a collective term for digital color spaces based on the separation of luma and chroma.

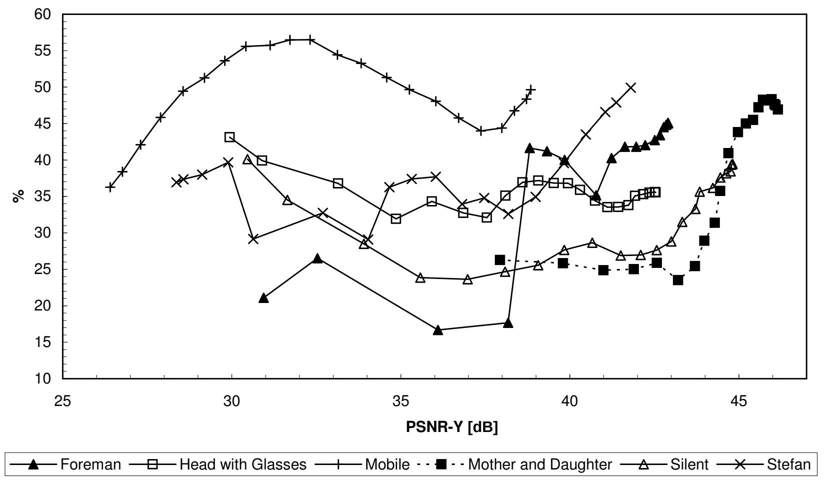


Figure 2.12: Bit rate savings of H.264/AVC AHM 2.0 relative to DivX 5.1, shown per measured quality value (in terms of Y-PSNR) and for all CIF sequences.

However, in most cases, it does correlate well with subjective quality perception. As such, it is a valid alternative for subjective tests that are typically difficult to set up.

In Figure 2.12, the bit rate savings of H.264/AVC AHM 2.0 relative to DivX 5.1 are shown per measured quality value (in terms of PSNR-Y) and for all CIF sequences (in the legend, Mobile is used as a short for the *Mobile and Calendar* test sequence). This figure makes clear that the bit rate savings are heavily dependent on the type of video content: sequences with similar characteristics do not even yield similar bit rate savings (e.g., Mobile and Calendar on the one hand and Stefan on the other hand). Significant bit rate savings are obtained for very low as well as for very high qualities. This confirms the fact that H.264/AVC is designed to support a wide range of bit rates.

The most remarkable aspect of Figure 2.12 is the fact that the bit rate savings for Mobile and Calendar are significantly higher than for the other sequences. Mobile and Calendar is the most complex video sequence used in the experiments; it has the notorious reputation of being a benchmark for source coding algorithms. However, many of the advanced features of H.264/AVC seem to encode Mobile and Calendar very efficiently.

In Figure 2.13, the average bit rate savings for each sequence are shown. Putting everything together, H.264/AVC AHM 2.0 achieves an average bit rate saving of 42% at QCIF resolution and 38% at CIF resolution, if the quality is measured in terms of PSNR. Although these numbers do not clearly confirm

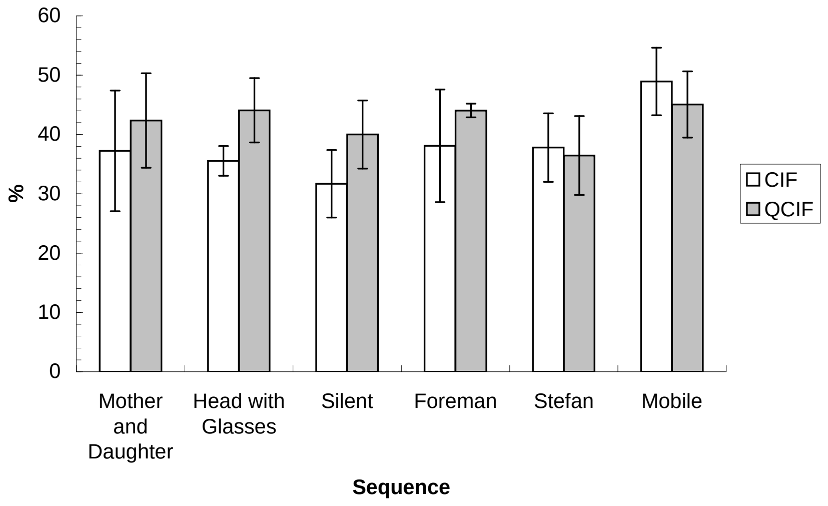


Figure 2.13: Average bit rate savings of H.264/AVC AHM 2.0 compared to DivX 5.1, shown per sequence (in terms of Y-PSNR). Error bars indicate the standard deviation.

Table 2.3: Average bit rate savings for video streaming applications [70].

	MPEG-4 Visual ASP	H.263 HLP	MPEG-2 Video MP
H.264/AVC MP	37.44%	47.58%	63.57%
MPEG-4 Visual ASP	-	16.65%	42.95%
H.263 HLP	-	-	30.61%

the goal of achieving 50% bit rate savings, it does show that significant savings can be obtained by making use of coding tools available in the H.264/AVC specification. For a more detailed overview of the results obtained, including the use of the Just Noticable Difference (JND) quality metric, we would like to refer the interested reader to [14] and to the dissertation of Peter Lambert [73].

Our observations are also in line with the RD performance results as reported by Wiegand *et al.* in [70]. A key result for video streaming applications is shown in Table 2.3, comparing the coding efficiency of the Main Profile (MP) of H.264/AVC to the Main Profile of H.262/MPEG-2 Video, the High-Latency Profile (HLP) of H.263, and the Advanced Simple Profile (ASP) of MPEG-4 Visual for a number of CIF video sequences. These results clearly indicate that H.264/AVC-compliant encoders may achieve essentially the same reproduction quality as encoders that are compliant with prior video coding standards, while typically allowing a reduction of the bit rate with 40 to 60%.

2.5.2 RD performance of H.264/AVC FRExt

In the first version of H.264/AVC, an integer-valued 4×4 transform is employed instead of the traditional, 8×8, floating-point Discrete Cosine Transform (DCT). This 4×4 transform, characterized by a low complexity on the one hand and an exact reversibility on the other hand, especially fits the needs of low bit rate and low resolution applications. In FRExt, as previously discussed, an 8×8 extension of the initial 4×4 transform is introduced, which is more suited for the coding of HD content due to a better preservation of fine details. This 8×8 transform is one of the key features of the important High Profile, the successor of the Main Profile. The High Profile is widely adopted by the industry; it is targeting high-end consumer video applications such as the efficient storage of HD content.

In [74], a number of performance data are reported that were obtained during a subjective quality evaluation done by the Blu-ray Disc Association. This test, based on the use of Mean Opinion Score (MOS) and conducted on 1920×1080 progressive scan film material at 24 frames per second, resulted in the following observations [59]:

- The High Profile of FRExt produced nominally (due to the use of MOS) better video quality than H.262/MPEG-2 Video when using only one-third as many bits (8 Mbps versus 24 Mbps);

- The High Profile of FRExt produced nominally transparent (i.e., difficult to distinguish from the original video without compression) video quality at only 16 Mbps.

Note that this test, as well as the experiments presented in the previous section, involved the use of very early implementations of H.264/MPEG-4 AVC encoders compared to, for example, rather mature H.262/MPEG-2 Video encoders. As such, these results can be considered a rather conservative estimate. Indeed, the H.264/AVC standard contains a lot of coding tools that improve the coding efficiency when used properly, but a lot of experimentation and development are still needed to assess how to take advantage of these features (without excessive encoding times). For instance, multiple reference pictures and generalized B slice coded pictures are new in commercial encoders, and there is still a lot of research needed to gain insight in their full potential for particular applications.

2.6 Conclusions and original contributions

H.264/AVC is the newest entry in a long series of standards for digital video coding. This specification has been developed by the Joint Video Team (JVT), a cooperation of the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). The main goals of this standard for general purpose video coding are an enhanced coding efficiency (targeting an average bit rate reduction of 50% compared to previous video coding standards, while maintaining a similar visual quality), the provision of a "network-friendly" video representation (aiming at mobile networks and the Internet), and a simple syntax specification (targeting simple and clean solutions that avoid the use of an excessive quantity of optional features and profile configurations).

In this chapter, we first outlined the history of the standardization process, describing the different versions of the H.264/AVC standard. Next, an overview was provided of the profiles that are defined in the H.264/AVC specification. The research further discussed in this dissertation is particularly focused on the first version of the H.264/AVC standard. This is mainly due to the time schedule of this work, which started in the Summer of 2002 when H.264/AVC was still known as the H.26L project.

Second, we provided an in-depth overview of a number of technical features of H.264/AVC, emphasizing the tools and concepts that are important for a good understanding of the remainder of this dissertation:

- the referencing mechanism between VCL NAL units and the different types of parameter sets;
- the Annex B syntax, used to separate NAL units in an elementary H.264/AVC bitstream;
- the use of start code prefix emulation prevention bytes;
- the employment of Exponential Golomb codes for the representation of high-level syntax elements;
- the use of Supplemental Enhancement Information, which is helpful for practical decoding or presentation purposes.

Finally, we reported the key results of a number of experiments that were conducted in the context of this research. These experiments compared the RD performance of an initial implementation of the first version of the H.264/AVC specification with the RD performance of a number of implementations of other coding formats. Combining our results with other achievements published in the scientific and technical literature, we learned that the coding tools of the new H.264/AVC design provide approximately a 50% bit rate savings for equivalent perceptual quality, relative to the performance of prior standards. This observation is especially true for higher-latency applications, which allow the use of more advanced motion estimation and motion compensation techniques.

Our contributions in the domain of H.264/AVC video coding can be found in the following publications. The second paper is an equal contribution among the first two authors.

1. Dieter Van Rijsselbergen, Wesley De Neve, Rik Van de Walle. GPU-driven Recombination and Transformation of YCoCg-R Video Samples. In *Proceedings of the Fourth IASTED International Conference on Circuits, Signals, and Systems (IASTED CSS 2006)*, pages 21–26, San Francisco, California, USA, November 2006.

2. Peter Lambert, Wesley De Neve, Philippe De Neve, Ingrid Moerman, Piet Demeester, Rik Van de Walle. Rate-Distortion Performance of H.264/AVC Compared to State-of-the-Art Video Codecs. In *IEEE Transactions on Circuits and Systems for Video Technology*, 16(1):134-140, January 2006.

3. Wesley De Neve, Dieter Van Rijsselbergen, Charles Hollemeersch, Jan De Cock, Stijn Notebaert, Rik Van de Walle. GPU-Assisted Decoding of Video Samples Represented in the YCoCg-R Color Space. In *Proceedings of the 13th ACM International Conference on Multimedia*, pages 447–450, Singapore, November 2005.

4. Wesley De Neve, Peter Lambert, Sam Lerouge, Rik Van de Walle. Assessment of the Compression Efficiency of the MPEG-4 AVC Specification. In *Proceedings of SPIE/Electronic Imaging 2004*, Volume 5308, pages 1082–1093, San Jose, California, USA, January 2004.

Chapter 3

Adaptivity provisions in H.264/AVC

No, there is basically no such limit. Actually I think there is a limit, but it is kind of big - about 4 trillion frames, which is about 4.5 years of video at 30 frames per second.

Gary J. Sullivan, co-chair of the JVT, regarding the maximum GOP size in an H.264/AVC bitstream.

3.1 Introduction

In contrast to prior specifications for digital video coding, H.264/AVC can be seen as a universal standard for the representation of moving pictures: its coding tools allow covering a wide spectrum of bit rates, ranging from 50 Kbps for mobile content (176×144, 10-15 Hz) to 8 Mbps for full high-definition and progressive video content (1080p or 1920×1080, 24 Hz), and beyond. In addition, taking into account the tools that offer support for an efficient network integration, error resilience, and bit rate adaptivity, it is likely that H.264/AVC bitstreams will be deployed in various usage environments.

In this chapter, we outline a number of adaptivity provisions that are available in the video coding layer of the H.264/AVC standard. In particular, four content adaptation tools are investigated that are mainly targeting the adaptation of pre-encoded content: bitstream switching, Region of Interest (ROI) coding, quality scalability, and temporal scalability. These adaptivity features may for instance be used to deal with bandwidth fluctuations, a typical characteristic of present-day best-effort network technologies [75].

During the discussion of the aforementioned adaptivity provisions in the H.264/AVC standard, the emphasis is put on the employment of temporal scalability in H.264/AVC bitstreams. Our motivation is twofold: first, this content adaptation technique is frequently used further in our research, and second, the implementation and exploitation of temporal scalability in H.264/AVC can be considered a topic that has not been extensively studied yet, as it is only covered by a rather limited number of scientific publications. Finally, before concluding this chapter, we also give a prospective overview of the adaptivity features that are offered by the scalable extensions to H.264/AVC.

3.2 Switching pictures for bitstream switching

3.2.1 Background

Video streaming has emerged as one of the most important applications over the Internet. It is also considered to be one of the killer applications for third generation wireless networks. However, the best-effort nature of current network technologies often causes variations in the amount of available bandwidth. To accommodate these bandwidth fluctuations, a server or intelligent gateway can scale the bit rate of the compressed video stream that is transmitted to a receiver.

For applications where online encoding is performed and where the encoder receives feedback on the available bandwidth of the transmission channel, bit rate adaptation can be achieved by dynamically changing the quantization parameter. However, when already encoded content is used, bandwidth adaptation may require the availability of more than one coded representation of the same content at different bit rates, or it may require the existence of non-reference pictures in the coded bitstream for the purpose of temporal scalability. The parallel provision of multiple representations of the same content is called simulstore in the context of a storage scenario, while it is referred to as simulcast in the context of multicast-based transmission.

When using simulstore, a server may dynamically switch between the different bitstreams to take into account the variations of the bandwidth available to the client. In prior video coding standards, perfect (mismatch-free) switching between bitstreams is only possible at pictures that do not use any information prior to their location, i.e. at I pictures[1]. Switching at P pictures causes error propagation due to the use of Motion-Compensated Prediction (MCP). The drawback of using I pictures is that these pictures require a much larger

[1]In the context of H.264/AVC, we define an I picture as a picture that entirely consists of I slices. P and B pictures are defined in a similar way.

number of bits than P pictures for the representation of content with the same fidelity (since I pictures do not exploit any temporal redundancy). As such, a lot of applications try to minimize the amount of I pictures in a bitstream. Consequently, these applications only allow to switch infrequently between different representations of the same content.

3.2.2 Switching pictures

As extensively discussed in [66] by Karczewicz and Kurceren, the Extended Profile of H.264/AVC includes a new feature consisting of two picture types, namely SP pictures and SI pictures, consisting entirely of SP slices and SI slices respectively. SP pictures and SI pictures make it possible for a decoder to switch between representations of the video content that use different data or picture rates. The coding method as defined for SP and SI pictures allows to obtain pictures that are having identical reconstructed sample values, even when different reference pictures are used for their prediction.

Similar to P pictures, SP pictures make use of MCP to exploit temporal redundancy in a sequence of pictures. The difference between SP and P pictures is that SP pictures allow identical pictures to be reconstructed, even when they are predicted using different reference pictures. Thanks to this property, SP pictures can be used instead of I pictures for the implementation of a feature such as bitstream switching. At the same time, since SP pictures - unlike I pictures - are using MCP, their representation requires significantly fewer bits than I pictures to achieve a similar perceptual quality.

Figure 3.1 and Figure 3.2 show how SP pictures can be used to switch between different bitstreams, respectively encoded at a different bit rate and temporal resolution. Within each encoded bitstream, SP pictures are placed at those locations at which switching from one bitstream to another one is allowed (pictures $\mathrm{S}_{1,n}$ and $\mathrm{S}_{2,n}$ in Figure 3.1). In what follows, such SP pictures are referred to as primary SP pictures.

Further, for each primary SP picture, a corresponding secondary SP picture is generated, which has the same identical reconstructed values as the primary SP picture. A secondary SP picture is sent only during bitstream switching. In Figure 3.1, the SP picture $\mathrm{SP}_{12,n}$ is the secondary representation of $\mathrm{S}_{2,n}$; it will be transmitted only when switching from bitstream 1 to bitstream 2. $\mathrm{S}_{2,n}$uses the previously reconstructed pictures from bitstream 2 as reference pictures, while $\mathrm{S}_{1,n}$ uses the previously reconstructed pictures from bitstream 1 as reference pictures. However, thanks to the particular encoding of the secondary SP picture, the reconstructed values of the pictures $\mathrm{S}_{2,n}$ and $\mathrm{SP}_{12,n}$ are identical.

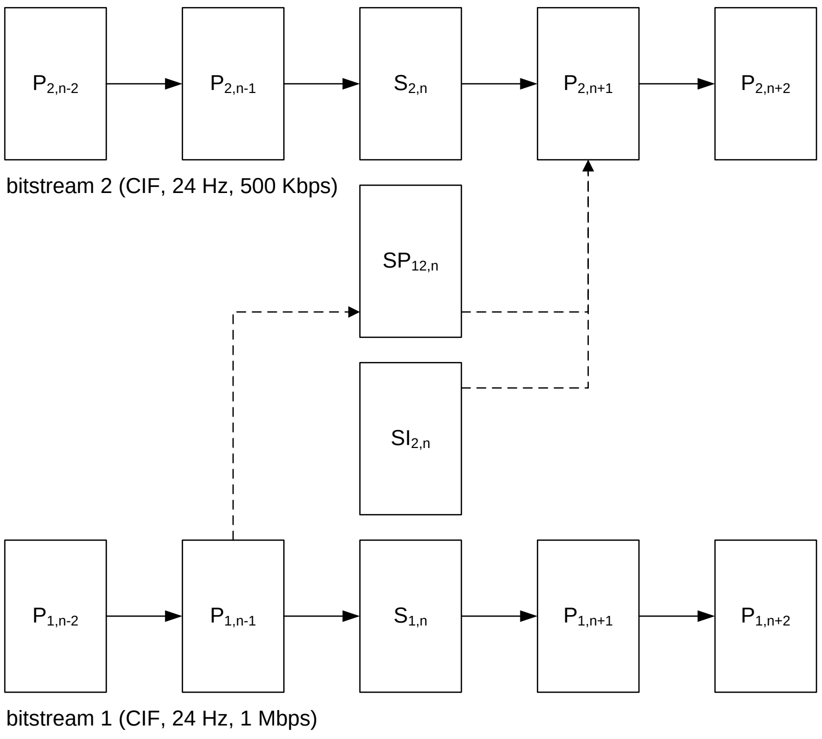


Figure 3.1: Use of SP and SI pictures for switching between bitstreams encoded at a different bit rate [66].

The encoding of secondary pictures is affected when switching occurs between bitstreams that are representations of different video sequences (e.g., bitstreams originating from different cameras capturing the same event but from different angles). Specifically, MCP of pictures from one bitstream using reference pictures from another bitstream where both bitstreams represent different sequences will not be as effective as when both bitstreams correspond to the same sequence. In such cases, using spatial prediction for the secondary SP pictures could be more efficient. This is also illustrated in Figure 3.1, where a secondary SP picture is denoted as $\mathrm{SI}_{2,n}$ to indicate that this is a picture encoded with SI slices, using spatial prediction and having identical reconstructed values as the corresponding secondary SP picture $\mathrm{SP}_{12,n}$ and the primary SP picture $\mathrm{S}_{2,n}$.

SI pictures can also provide random access points to the bitstream, and have, together with SP pictures, further implications regarding error recovery and error resilience [66]. For example, multiple representations of a single

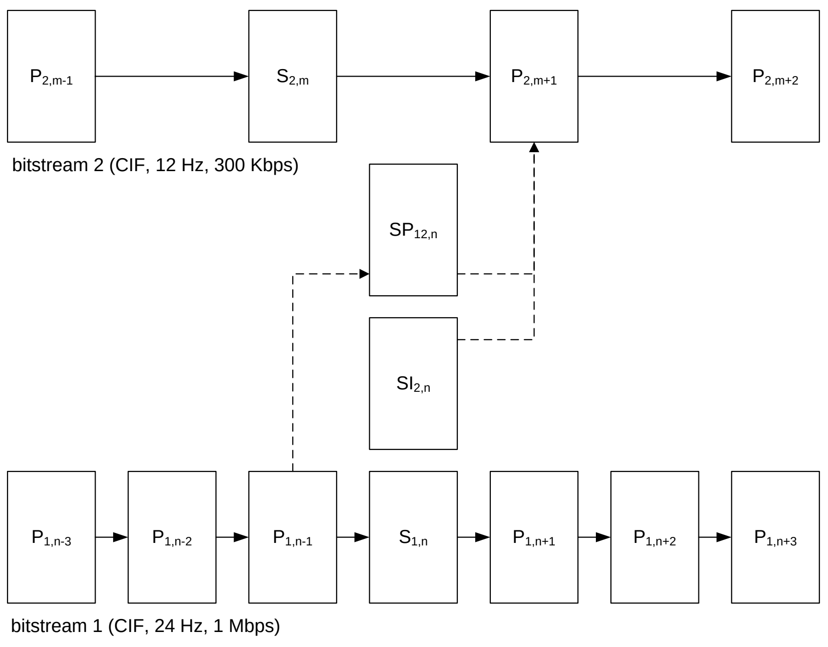


Figure 3.2: Use of SP and SI pictures for switching between bitstreams encoded at a different picture rate.

picture in the form of SP pictures predicted from different reference pictures, e.g. the immediate previously reconstructed picture and a reconstructed picture further back in time, can be used to increase error resilience and/or error recovery. Indeed, consider the case in which an already encoded bitstream is streamed to a client and where packet loss has led to the disposal of a picture or a slice. The client signals the lost picture to the server, which then responds by sending one of the secondary representations of the next SP picture. This secondary representation uses the reference pictures that have been correctly received by the client, making it possible to prevent or stop error propagation.

To conclude, the use of switching pictures introduces an additional storage cost at the server. However, their use makes it possible to reduce the bandwidth consumption in a network while still allowing features such as random access and fast switching between bitstreams with different characteristics.

3.3 FMO for region of interest coding

As previously explained in Chapter 2, the error resilience aspect of slices can be enhanced through the use of Flexible Macroblock Ordering (FMO). This technique modifies the way pictures are partitioned into slices and macroblocks by relying on the concept of slice groups.

A slice group is a subset of the macroblocks in a coded picture and may contain one or more slices. Within each slice in a slice group, macroblocks are coded in raster scan order. Multiple slice groups allow mapping the sequence of coded macroblocks to the decoded picture in a number of flexible ways. The allocation of the macroblocks is determined by a macroblock to slice group map that indicates to which slice group each macroblock belongs. This map is conveyed by a PPS.

FMO allows to split a picture into a number of macroblock scanning patterns such as interleaved slices (FMO type 0), a checker-board type of mapping (FMO type 1), and one or more rectangular *foreground* slice groups and a *background* or left-over slice group (FMO type 2). FMO type 1 is useful for error concealment in video conferencing applications, when the different slice groups are transmitted in separate packets and when one of the packets gets lost [15]. Thanks to the organization of the video data, the samples of a missing slice are surrounded by samples of correctly received and decoded slices. Consequently, the correctly decoded samples can be used for the reconstruction of the missing samples.

In Figure 3.3, we illustrate the positive impact of FMO type 1 on the objective video quality in case of uniform packet loss. The following test setup was used. The Stefan test sequence was extended by pasting it six times in a row sequentially, generating a CIF video sequence with a duration of 60 seconds and with a frame rate of 30 Hz. This extended sequence was subsequently encoded by making use of a slice size of 50 macroblocks; a GOP length of 18 pictures; a GOP structure that takes the form of IBBP...; and QP values of 40/40/42 and 28/28/30 (for I/P/B slices). The chosen QP values represent a low and a high quality, respectively. The encoding was done once without FMO and once with FMO type 1 using four slice groups (every slice group contains two slices). Also, the first picture of the sequence was intra-coded to compensate for the artificial scene cut. Furthermore, CABAC was used as entropy coding scheme. If the decoder detects a missing NAL unit, the macroblocks of the corresponding slice are reconstructed using a spatial interpolation algorithm provided by the H.264/AVC reference software.

The abbreviation HQ (resp. LQ) in the legend of Figure 3.3 stands for high quality (resp. low quality), which means that a low (resp. high) quantization

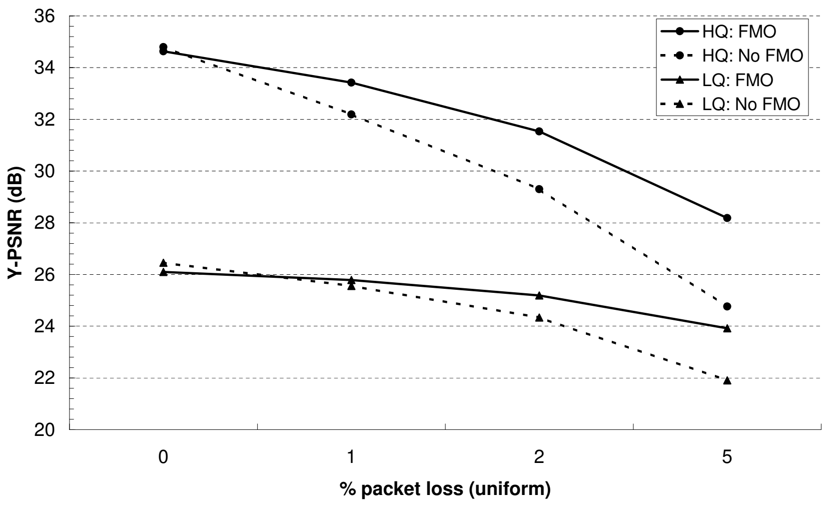


Figure 3.3: Decrease in objective quality due to uniform packet loss.

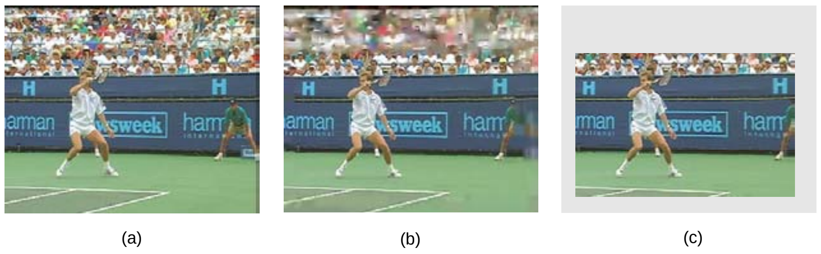


Figure 3.4: Example of a tennis game with ROI encoding and adaptation [68].

parameter was used (see above). In Figure 3.3, we observe that the difference in average Y-PSNR between the cases with and without FMO becomes bigger when the percentage of uniform packet loss rises. The differences in Y-PSNR are also bigger when the bitstreams are encoded with high quality. The gains in Y-PSNR range from 0.2 dB to 3.4 dB with an average of 1.9 dB. For a more detailed discussion regarding the use of FMO as an error resilience tool, including an analysis of its cost in terms of coding efficiency and computational complexity, we would like to refer the interested reader to [15], [76], and [77].

Besides error resilience, FMO can be used for other purposes as well. For example, FMO type 2 has been demonstrated useful for employment in ROI type of applications, allowing to prioritize particular slice groups. Indeed, a background slice group can be encoded at a lower quality than a more impor-

Table 3.1: Achieved bit rate savings by altering the QP inside and outside the ROI for the Stefan test sequence (in Kbps) [68].

		QP region of interest					
		8	16	24	32	40	48
	8	11570	9385	8070	7467	7288	7230
	16	8026	5836	4521	3919	3740	3682
QP background	24	5892	3702	2373	1770	1585	1528
	32	4913	2719	1384	757	565	508
	40	4646	2448	1118	483	267	204
	48	4591	2396	1055	418	191	118

tant foreground slice group. This scenario is illustrated in Figure 3.4: in (a) the entire video sequence is transmitted at full quality, while in (b) the background is transmitted at a lower quality to save bandwidth. Finally, in (c) the entire background is dropped in favor of the more important foreground (ROI extraction). However, the removal of slice groups from a coded bitstream is not allowed by the first version of the H.264/AVC standard. In Chapter 6, we will outline a workaround for this restriction, based on the use of skipped slices.

FMO type 2, which allows specifying rectangular areas of interest in a picture, can be used without much overhead compared to bitstreams without FMO. The cost was less than 1% in most of the experiments described in [68]. Table 3.1 illustrates the variation in bit rate that can be achieved by altering the QP inside and outside the ROI for the well-known Stefan test sequence (a screenshot of this test sequence is shown in Figure 3.4). By comparing the bit rates within a column, one can observe that the bit rate can be significantly affected by changing the visual quality of the background. For instance, by taking a QP of 24 for the entire Stefan sequence, which corresponds to a good quality, the bit rate drops with over 50% if the quantization parameter of the background is set to 48. By comparing the numbers of Table 3.1 within a row, one similarly sees how the bit rate changes if the quantization parameter within the ROI is altered, e.g. by raising the quality of the ROI.

3.4 Data partitioning for SNR scalability

Some coded information (e.g., motion vectors and other prediction information) is more important than other information for the purpose of representing the video content. Therefore, H.264/AVC allows the syntax of each slice to be separated into different partitions for transmission, depending on a categorization of the syntax elements. This feature is known as data partitioning. Its main purpose is to be used as an error resilience tool for the transmission of coded video over channels that allow selective protection of the different partitions (Unequal Error Protection; UEP) [75]. As such, data partitioning allows enhancing data loss robustness without harming coding efficiency.

When data partitioning is in use, the coded data of each slice is split up into three partitions, and each partition is put into a separate NAL unit (each having a different NAL unit type). The following partitions are defined by the H.264/AVC specification:

- **partition A** contains the slice header, macroblock types, quantization parameters, prediction modes, and motion vectors;
- **partition B** contains the residual data of intra-coded macroblocks;
- **partition C** contains the residual data of inter-coded macroblocks.

Obviously, the loss of individual partitions still results in error propagation. The information contained in partitions B and C is useless for error resilience purposes at the decoder if the corresponding partition A is not present. However, even when partitions B and C are missing, the information in partition A can still be used for error concealment purposes at the decoder. Consequently, partition A is independent of partitions B and C. In [18], we show that partition B is independent of partition C when constrained intra prediction is in use. Constrained intra prediction restricts an encoder to only use information from other intra-coded macroblocks for intra prediction on the one hand and for entropy coding using CAVLC on the other hand.

Data partitioning, although not a true form of scalable coding, also provides means for partitioning the coded video data into a number of priority classes: essential data and additional data. This simple and low penalty technique[2] for generating layers of data can be exploited to reduce the bit rate of a bitstream for a particular usage environment, offering a coarse but efficient method for quality scalability[3].

In Table 3.2, we report a number of performance data when using data partitioning for the implementation of quality scalability in the first version of the H.264/AVC standard, coping with bandwidth fluctuations in video conferencing applications. The results are obtained for the Silent test sequence (QCIF; 15 Hz), encoded at 256 Kbps and with an IDR period of 25 pictures. Data

[2]The overhead of data partitioning is in general less than 1% of the total bit rate [78].

[3]Note that the adapted bitsteams are no longer compliant with the H.264/AVC standard.

Table 3.2: Selected bit rates and PSNR values when using data partitioning for SNR scalability, targeting video conferencing applications. The achieved bit rates are expressed in Kbps while the luma PSNR is given in terms of decibels (dB).

partition	IMBR 0			IMBR 5			IMBR 10		
	rate	PSNR	cost[a]	rate	PSNR	cost	rate	PSNR	cost
A & B & C	256.4	45.1	-	256.3	43.0	-	256.4	41.2	-
A & B	113.8	31.6	38.6	151.7	32.8	38.7	177.3	33.8	38.4
A	97.5	26.6	37.6	94.4	26.1	35.3	93.2	25.9	33.7

[a]Cost refers to the PSNR obtained for the sequence, encoded using all partitions at the lower rates given in the table. It shows the ‘price’ of providing SNR scalability using data partitioning.

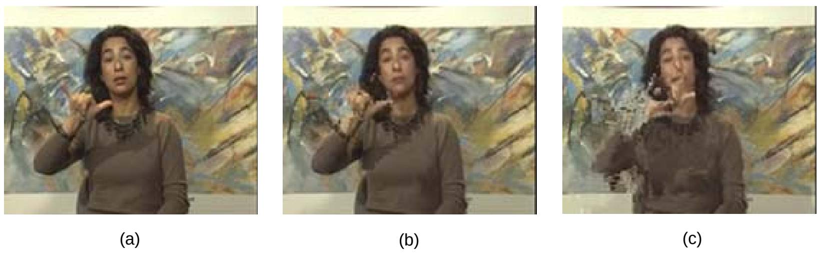


Figure 3.5: Example of SNR scalability using data partitioning, shown for frame 75 of the Silent test sequence: (a) no data partitioning; (b) disposal of partition C; and (c) disposal of partition B and C (showing ghosting artifacts). A value of 5 is used for the IMBR parameter.

partitioning and constrained intra prediction were enabled, as well as RD optimization. Missing information, due to the removal of certain partitions, was reconstructed as follows [18]: for a missing C partition, the residual information for inter-coded macroblocks is assumed to be zero. When the B partition is missing as well, intra-coded macroblocks are copied from the previous frame. Inter-coded macroblocks are still decoded as usual (i.e., using MCP), except for the residue, which is assumed to be zero.

Also, Random Intra Macroblock Refresh (IMBR) was used, which forces an encoder to insert a given number of intra-coded macroblocks in P or B slice coded pictures. For example, when IMBR is set equal to 10, besides the macroblocks for which the mode decision algorithm had decided to use intra prediction, at least 10 randomly chosen macroblocks per picture will be intra-coded as well. A number of representative screenshots are provided in Figure 3.5. For more comprehensive results regarding the use of data partitioning in general and for the implementation of SNR scalability in particular, we would like to refer the interested reader to [78] and [18], respectively.

3.5 Multi-layered temporal scalability

3.5.1 Background

Digital video coding involves the accurate reproduction of color and the fluent representation of motion. Temporal scalability, which is also known as picture dropping or bitstream thinning, influences the smoothness of the motion representation: it refers to the ability to remove some coded pictures from a bitstream while still obtaining a decodable remaining sequence of pictures.

Temporal scalability is typically implemented by organizing the coded pictures in multiple layers. By combining these layers, the full temporal resolution can be offered, as available in the original video sequence. Since the input picture rate is partitioned between the base layer and the different enhancement layers, a decoder does not need to be much more complex than a single-layered decoder in order to support temporal scalability features. This is for instance in contrast to a decoder which needs to support spatial scalability features; its design typically includes additional logic for the purpose of upsampling, et cetera.

Several applications may benefit from a tool such as temporal scalability [79].

- A first application is rate shaping, which has the objective to match the bit rate of a coded video bitstream to the target rate constraint. Many networks are not capable of providing channels with a constant throughput. Thus, many streaming servers monitor the network conditions and adjust the amount of transmitted data accordingly. As temporal scalability offers flexibility in bandwidth partitioning between different layers, one way to control the bit rate of the transmitted bitstream is to decide which temporal enhancement layers can be transmitted on top of the base layer.

- A second application of temporal scalability is UEP. In order to apply UEP, video bitstreams have to be organized in portions of different importance of visual quality. Techniques achieving this goal include data partitioning and scalable or layered coding.

- Temporal scalability also enables lowering the complexity of the decoding process. This is for instance useful for applications requiring a transition from high to low temporal resolution progressive video formats (e.g., from 720p content encoded at 60 Hz to 720p content encoded at 30 Hz).

In the remainder of this section, which partly follows the structure of [80], we first explain how temporal scalability is realized in traditional video coding formats. Next, the format-independent concepts of sub-sequences and sub-sequence layers are introduced. Finally, we discuss into detail how the concept of sub-sequences and sub-sequence layers can be employed in H.264/AVC.

3.5.2 Conventional temporal scalability

Individually disposable pictures

In video coding formats such as MPEG-1 Video and H.262/MPEG-2 Video, bi-directionally coded pictures (B pictures) provide an inherent form of temporal scalability. These pictures are placed outside the decoding loop, allowing shortcuts to be taken during the decoding process without causing drift or long-term visual artifacts. A similar functionality can be achieved in H.264/AVC by making use of non-reference B pictures, i.e. pictures that only consist of non-reference B slices.

The identification of non-reference B pictures in H.264/AVC requires checking the value of the `nal_ref_idc` syntax element in the NAL unit header[4] and the value of the `slice_type` syntax element in the slice header syntax structure. Note that two values may be used for signaling the property that a slice is bi-directionally coded: a value of one for `slice_type` implies that other slice types may occur in the coded picture the slice belongs to, while a value of six implies that all other slices are bi-directionally coded as well (a similar remark can be made for the other slice types).

The detection of all B slices in a bitstream implies that all slices need to be checked. As such, the complexity of this process is determined by the number of slices in a bitstream, and not by the number of pictures. The computational complexity of this analysis step may be simplified using an access unit delimiter NAL unit. Such a NAL unit signals the boundaries of an access unit and it also conveys the type of the primary picture in the access unit. However, this syntax feature is not used in practice at the time of writing. Similar functionality may be provided using SEI messages (see below).

The enhanced reference picture selection mode (Annex U) of H.263 allows signaling whether a particular picture is used as a reference picture, i.e. whether this picture is used for the prediction and reconstruction of other pictures. Consequently, a picture that is not used for prediction (a non-reference picture) can be safely disposed. The H.264/AVC syntax also includes a signaling mechanism to distinguish between reference and non-reference pictures, independent of the coding type of the slices constituting these pictures - this is, by means of the `nal_ref_idc` syntax element in the NAL unit header.

[4]`nal_ref_idc` equal to zero for a NAL unit containing a slice or slice data partition indicates that the slice or slice data partition is part of a non-reference picture.

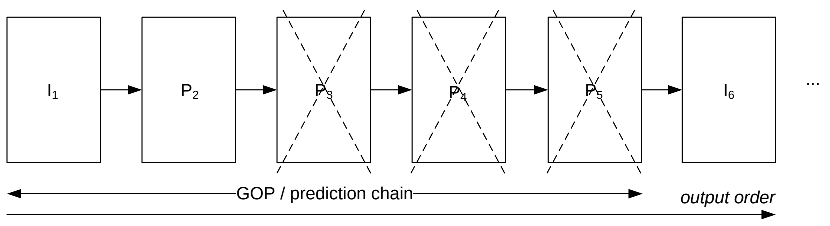


Figure 3.6: Disposal of picture chains.

Implementing temporal scalability by relying on `nal_ref_idc` results in a limited expressive power. This may already be sufficient for some applications. However, more expressive power is needed when layered bitstreams are employed. These bitstreams are typically constructed using inter-layer prediction mechanisms to improve the coding efficiency: typically, only the pictures in the topmost enhancement layer are then communicated as non-reference pictures to a decoder or a bitstream extractor (see below).

Disposal of picture chains

A known method to deal with a drastically dropped network bandwidth is to transmit intra-coded pictures only. When the network throughput is restored, inter-coded pictures can be transmitted again from the beginning of the next GOP. Generally, any chain of inter-coded pictures can be disposed safely if no other pictures are predicted from them. This observation can be utilized to treat inter-coded pictures at the end of a prediction chain less important than other inter-coded pictures (in coding order). This is illustrated in Figure 3.6, showing a typical coding pattern as it may be used in an MPEG-4 Visual bitstream for the purpose of video conferencing applications. A picture chain consisting of the inter-coded pictures P_3, P_4, and P_5 is discarded to cope with bandwidth congestion (the picture index denotes the output order). It is clear that the flexible design of H.264/AVC allows realizing such bandwidth control as well.

3.5.3 Sub-sequences and sub-sequence layers

Definition

Sub-sequences and sub-sequence layers are defined independently of the coding format used. A sub-sequence [80] represents a number of inter-dependent pictures that can be disposed without affecting the decoding of any other sub-sequence in the same sub-sequence layer or any sub-sequence in any lower

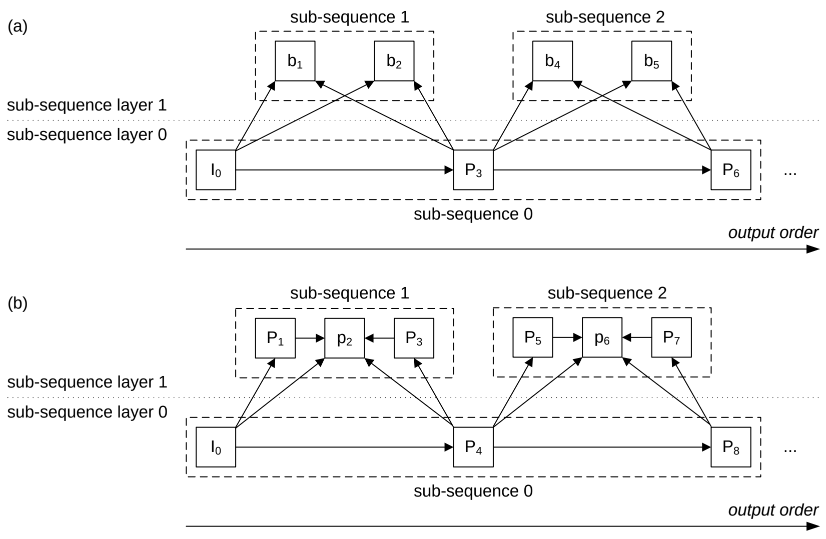


Figure 3.7: Example of sub-sequences: (a) coding pattern IbbPbbP…; (b) coding pattern IPpPPPpPP…

sub-sequence layer. Sub-sequences were introduced in [81] by Miska Hannuksela as the enhanced concept of a GOP. Indeed, in contrast to a GOP, a sub-sequence does not have to start with an intra-coded picture. Furthermore, arbitrary temporal prediction structures are allowed within a sub-sequence to improve the coding efficiency.

A sub-sequence is a set of coded pictures within a sub-sequence layer. Such a sub-sequence layer contains in its turn a subset of the coded pictures in a video sequence. The layers can be ordered hierarchically based on their dependency of one another. The base layer is independently decodable. The first enhancement layer depends on some of the data in the base layer. The second enhancement layer depends on some of the data in the first enhancement layer and in the base layer and so on.

Use of sub-sequences and sub-sequence layers

Pictures in a coded bitstream can be assigned to sub-sequences and sub-sequence layers in numerous ways, provided that the coding structure used fulfills the requirements for dependencies between sub-sequences and sub-sequence layers: each picture belongs to exactly one sub-sequence, and each sub-sequence belongs to exactly one sub-sequence layer. Two possible sub-sequence configurations are visualized in Figure 3.7. A capital letter denotes a reference picture; a small letter indicates a non-reference picture. The second coding structure is adopted from [80][5]. The picture index denotes the output order.

Sub-sequence layers are numbered with non-negative integers. Number zero denotes the base layer. The first enhancement layer is associated with number one and each additional enhancement layer increments the layer number by one. A sub-sequence identifier is assigned to sub-sequences. Consecutive sub-sequences (within the same sub-sequence layer) should not have the same identifier, but no other numbering rule for sub-sequence identifiers is necessary.

Only entire sub-sequences and non-reference pictures should be disposed. When disposing a sub-sequence, any sub-sequence depending on the disposed one should be discarded as well. As such, the result of the disposal of a sub-sequence or sub-sequence layer is a valid bitstream. The decoding process for the remaining bitstream and the decoded picture buffer handling in particular is such that it does not depend on the presence or absence of any disposable sub-sequences or sub-sequence layers.

Since a sub-sequence in the base layer can be decoded independently, the start of a base layer sub-sequence can be used as a random access position (closed GOP). Implementations of fast forward functionality can for instance make use of this property. Further, the subjective quality, in terms of motion smoothness, increases along with the number of decoded enhancement layers.

3.5.4 The sub-sequence concept in H.264/AVC

The flexible design of the H.264/AVC standard introduces a number of novelties that make the achievement of temporal scalability in this coding format more challenging than in its predecessors.

1. The fundamental unit of processing in the VCL of H.264/AVC is a slice. Consequently, the H.264/AVC specification only defines slice types, and not pictures types. As such, stricto senso, a B picture for instance does not exist in the H.264/AVC standard.

2. Any coded picture other than an IDR picture may contain I, P, or B slices in any combination (dependent on the profile used).

3. Whether or not a picture is used as a reference picture is indicated independently from the signaling of the slice types, which implies that a B slice coded picture can for instance be stored as a reference picture.

[5]In the context of H.264/AVC, the first coding pattern is compliant with its Main Profile, while the second coding structure is compliant with the Baseline Profile.

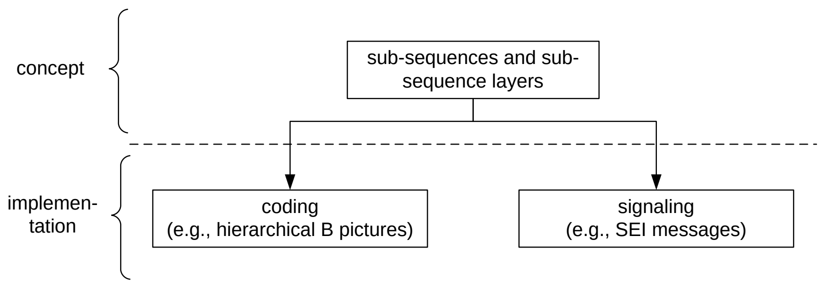

concept
sub-sequences and sub-sequence layers
implemen-tation
coding (e.g., hierarchical B pictures)
signaling (e.g., SEI messages)

Figure 3.8: Schematic overview of the use of sub-sequences and sub-sequence layers in H.264/AVC.

4. The decoding order of pictures is completely decoupled from their output order, which for instance implies that a B slice can have all of its references in the past.

5. P and B slices allow the use of multiple reference pictures for the prediction of sample values. P slices can use at most one prediction signal for a block of samples, whereas B slices can use at most two prediction signals for the prediction of a block of samples.

Most of the aforementioned features aim at enabling complex prediction structures in order to achieve a maximum coding efficiency. However, the temporal dependencies introduced by these complex prediction structures may significantly reduce the number of adaptation steps that can be performed along the temporal axis of H.264/AVC bitstreams. To allow the use of complex prediction strategies on the one hand and to enable temporal adaptation features on the other hand, sub-sequences and sub-sequence layers can be created and signaled in H.264/AVC bitstreams.

Before describing the creation and signaling of sub-sequences and sub-sequence layers in H.264/AVC, the semantics of two syntax elements are discussed that are important in the context of the disposal of reference pictures. This discussion is based on [80] and [82], as well as on technical discussions conducted on the *JVT Experts*[6] and *MP4-Tech*[7] reflectors. Finally, provided as a hold, Figure 3.8 gives a schematic overview of the practical use of the sub-sequence and sub-sequence layer concepts in the context of the H.264/AVC video coding format.

[6] `http://mailman.rwth-aachen.de/mailman/listinfo/jvt-experts`
[7] `http://www.m4if.org/public/publiclistreg.php`

Semantics of frame_num

The frame_num syntax element in the slice header syntax structure is used to keep track of the reference pictures in an H.264/AVC bitstream. The behavior of this syntax element depends on whether a picture is used as a reference picture or not, which is communicated by the nal_ref_idc syntax element in the header of a NAL unit: frame_num acts as a counter that is incremented after the decoding of a reference picture[8]. This simple process enables a decoder to detect the loss of one or more reference pictures and to possibly apply error concealment techniques without losing track of what is going on.

Since the proper decoding of a non-reference picture is not necessary for the proper decoding of subsequent pictures, frame_num was designed so that a missing non-reference picture would not cause frame_num to indicate the presence of a problem when a non-reference picture is missing (i.e., frame_num is not incremented after the decoding of a non-reference picture). The value of frame_num is reset to zero whenever a new coded video sequence begins, as well as when this counter reaches its maximum. This maximum is indicated in the active SPS by log2_max_frame_num_minus4.

The frame_num parameter is primarily a loss robustness feature: it provides a form of robustness to transmission-channel losses of reference pictures. However, besides error concealment purposes, frame_num can also be used for the purpose of content adaptation, e.g. to assist in the identification of particular temporal enhancement layers. Chapter 5 illustrates such functionality.

Semantics of gaps_in_frame_num_value_allowed_flag

The gaps_in_frame_num_value_allowed_flag syntax element in an SPS informs a decoder whether gaps in the frame_num syntax element are allowed or not - this is, whether the loss of reference pictures is intentional or not. Unexpected loss of reference pictures can for instance be due to packet loss in a congested network. Reference pictures can also be disposed intentionally by a streaming server or a gateway.

- A value of one for gaps_in_frame_num_value_allowed_flag in the active SPS signals to a decoder that intentional disposal of reference pictures is allowed. In this context, decoders shall not infer an unintentional picture loss immediately when a missing picture number is detected but will rather update the decoded picture buffer so that the default index order remains the same as in the encoder. Otherwise, incorrect reference pictures may be used for the reconstruction of remaining

[8]A better name for frame_num would probably have been ref_pic_num.

pictures. An update of the decoded picture buffer is realized by inserting a so-called “non-existing” picture in this buffer, as if the pictures with an absent value for `frame_num` were received and decoded normally. Unexpected pictures losses can still be deduced when “non-existing” reference pictures are referred to in the subsequent decoding process.

- A value of zero for `gaps_in_frame_num_value_allowed_flag` in the active SPS informs a decoder that reference pictures may not be missing. When a decoder detects a lost reference picture (by monitoring the continuity of the values of `frame_num`), it may invoke an error concealment process, and it may subsequently insert an error-concealed picture into the decoded picture buffer. Such concealment may be conducted by copying the closest temporally preceding picture that is available in the decoded picture buffer into the position of the missing picture. Decoders shall infer an accidental picture loss if any “invalid” picture is referred to in motion compensation.

This definition provides a well-defined decoder behavior for environments in which the system operation may result in the loss of reference pictures. For such a case, it might not really be necessary to specify how the decoder would respond to a picture loss - however, this feature allows the encoder to decide whether the decoder should interpret the loss of a picture as a real problem or not (by setting the value of `gaps_in_frame_num_value_allowed_flag` to zero or one). Further, it allows the encoder to understand how the decoder will respond to such events, which can be beneficial to know for various purposes (e.g., motion compensation using an error-concealed reference picture).

Note that the way the decoder responds to a missing reference picture when `gaps_in_frame_num_value_allowed_flag` is equal to one may often also be a good way for the decoder to respond to a missing reference picture when `gaps_in_frame_num_value_allowed_flag` is equal to zero. However, when `gaps_in_frame_num_value_allowed_flag` is equal to zero, it is at the discretion of the individual decoder designer to decide how the decoder should respond to missing reference pictures.

Coding of sub-sequences

In contrast to previous standards for video coding, the coding and output order of pictures in H.264/AVC is completely decoupled. Also, any picture can be marked as a reference picture and can be used for the prediction of subsequent pictures independent of the corresponding slice types. These novel features in

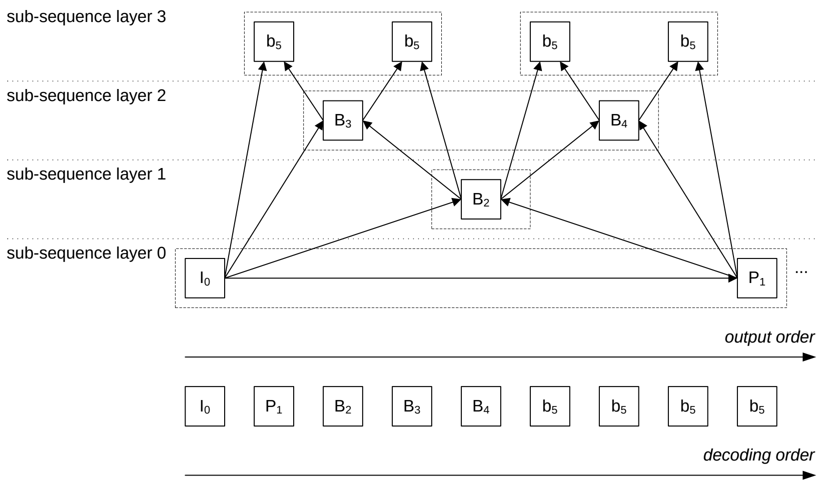


Figure 3.9: Hierarchical coding pattern with four temporal levels. Each picture is tagged with the value of the `frame_num` syntax element.

H.264/AVC, among other ones, allow the creation of arbitrary prediction structures, which are not supported by prior video coding standards. This flexibility in terms of possible coding structures makes it possible to organize the pictures in a bitstream to sub-sequences and sub-sequence layers in multiple ways.

Sub-sequences in H.264/AVC are typically created by relying on a hierarchical coding pattern. This is a coding structure in which the use of reordering between picture decoding order and picture output order takes the form of building up a coarse-to-fine structuring of temporal dependencies. For example, as shown in Figure 3.9, there might be a first conceptual layer consisting of an I picture followed by P pictures at a very low picture rate. Then a second conceptual layer can be created that consists of B pictures that are inserted between the pictures of the first layer in output order, using the pictures of the first layer (and possibly some preceding pictures of the second layer also) as references for the decoding process. It is easy to see that this technique can be repeated for an arbitrary number of enhancement layers, resulting in multi-layered temporal scalability. The pictures of the various layers are interleaved in the bitstream in a manner ensuring that all references needed by each picture are found at earlier positions in decoding order and ensuring that the total amount of decoded picture buffer memory does not exceed the capacity specified for the given profile and level.

B pictures are often used to implement hierarchical coding patterns [83, 84]. These B pictures are usually referred to as hierarchical B pictures and the

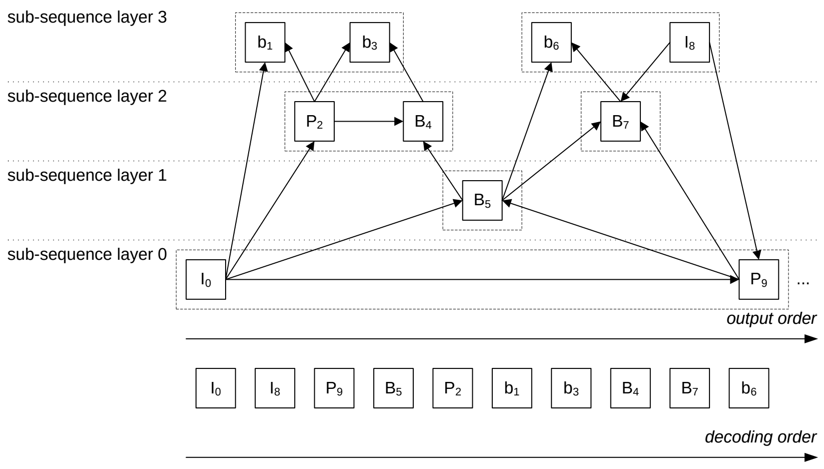


Figure 3.10: Explicit coding pattern (note that the representation of a picture may consist of multiple slices, where each slice has a different coding type).

resulting coding technique as pyramid coding. However, if coding efficiency is less important than encoding and decoding complexity, hierarchical I or P pictures can be used as well, or a mix of the different types. Such coding structures are referred to as explicit coding patterns. An example coding pattern, using hierarchical B pictures and offering four temporal levels, is shown in Figure 3.9. Figure 3.10 visualizes an explicit coding structure. Dashed boxes are used to denote the sub-sequences. The picture index denotes the display order.

The pattern shown in Figure 3.9 is an example of a dyadic, hierarchical prediction structure with four different layers[9]. The use of hierarchical coding structures is not restricted to the dyadic case. Furthermore, the prediction structure can be adjusted over time. Also, the concept of multiple reference pictures can be combined with hierarchical coding structures. In Figure 3.9, only neighboring pictures of a coarser or the same temporal level are used for MCP. Note that a hierarchical coding pattern is typically a good structure in terms of coding efficiency, but not in terms of end-to-end delay, unless MCP is restricted from using pictures as a reference that are located in the future [84].

For more details regarding the use of sub-sequences in the H.264/AVC video coding format, we would like to refer the interested reader to [80]. In this paper, the authors demonstrate that the temporal scalability features of

[9]Dyadic means that the number of pictures in every temporal enhancement layer is equal to the number of pictures residing in the lower layers, minus one. Consequently, the removal of an enhancement layer results in halving the picture rate.

Table 3.3: Sub-sequence information SEI message syntax [55].

sub_seq_info(payloadSize) {	**C**	**Descriptor**
sub_seq_layer_num	5	ue(v)
sub_seq_id	5	ue(v)
first_ref_pic_flag	5	u(1)
leading_non_ref_pic_flag	5	u(1)
last_pic_flag	5	u(1)
sub_seq_frame_num_flag	5	u(1)
if(sub_seq_frame_num_flag)		
sub_seq_frame_num	5	ue(v)
}		

H.264/AVC allow for a wider range of bit rate scaling compared to the more traditional techniques for picture dropping, without having to sacrifice coding efficiency. This is thanks to the flexible picture referencing mechanism of H.264/AVC, which makes it possible to efficiently reduce the temporal correlation in and between the temporal enhancement layers.

Signaling of sub-sequences

One of the design goals of the sub-sequence feature was to make the decoding process unaware of the hierarchical nature of a bitstream. Consequently, no layer number is added to the bitstream syntax or plays a role in the decoding process. In other words, the introduction of sub-sequences and sub-sequence layers required no changes to be made to the VCL syntax of H.264/AVC.

The use of SEI messages is the recommended technique to inform a bitstream extractor or decoder about the hierarchical coding structure of a bitstream, which is expressed in terms of sub-sequences and sub-sequence layers. Three types of SEI messages are defined for sub-sequences [55].

- The sub-sequence information SEI message[10], of which the syntax is shown in Table 3.3, is used to indicate the position of its associated picture in the data dependency hierarchy that consists of sub-sequence layers and sub-sequences. As such, this message maps a coded picture to a certain sub-sequence (by means of the `sub_seq_id` syntax element) and sub-sequence layer (by means of the `sub_seq_layer_num` syntax element). The use of sub-sequence information SEI messages is illustrated by Figure 3.11. A label $C_{i,j}$ has the following meaning: C denotes

[10]Sub-sequence information SEI messages shall not be present unless `gaps_in_frame_num_value_allowed_flag` in the SPS referenced by the picture associated with the sub-sequence information SEI message is equal to one.

the coding type of a slice, i indicates the value of `frame_num`, and j represents the top-down order of the slice within a picture. A label S_k has the following meaning: S denotes a sub-sequence information SEI message, while k represents the sub-sequence layer identifier[11].

- The sub-sequence layer characteristics SEI message and the sub-sequence characteristics SEI message are two SEI messages that provide statistical information (e.g., bit rate) on the indicated sub-sequence layer and sub-sequence respectively. Furthermore, the dependencies between sub-sequences can be indicated in the sub-sequence characteristics SEI message. For instance, this SEI message may contain a list of sub-sequences needed for the reconstruction of the pictures in the sub-sequence that is the target of the sub-sequence characteristics SEI message in question.

Decoders can use the sub-sequence-related SEI messages to reduce the complexity of the decoding process in case of a lack of computational resources. Also, these SEI messages provide a low-complexity solution for, among other use cases, streaming servers to dispose certain bitstream segments to meet bandwidth constraints. Indeed, the subsequence-related SEI messages abstract the bitstream syntax: the streaming server only needs to have knowledge about the NAL syntax of the H.264/AVC coding format, and not about the VCL syntax to discard particular bitstream segments.

Decoders may also detect in which sub-sequences and sub-sequence layers accidentally lost pictures reside (e.g., due to transmission errors), thus improving error resilience. This is done by monitoring the continuity of the `sub_seq_frame_num` syntax element, which is optionally conveyed by a sub-sequence information SEI message (see Table 3.3). The `sub_seq_frame_num` syntax element has similar semantics as `frame_num`: it acts as a counter that is incremented when the picture associated with the sub-sequence information SEI message is used as a reference. As such, gaps in `sub_seq_frame_num` signal the unintentional loss of a reference picture in a sub-sequence. The main difference with the `frame_num` syntax element lies in the fact that `sub_seq_frame_num` is defined within the context of a sub-sequence and not within the context of a coded video sequence[12].

[11] For the sake of completeness, it is noteworthy that Memory Management Control Operation (MMCO) commands may only occur in the base layer. These instructions are used to adjust the default operation of the reference lists in a decoder.

[12] `sub_seq_frame_num` is reset to zero whenever a new sub-sequence begins. This is in contrast to `frame_num`: its value is reset whenever a new coded video sequence starts.

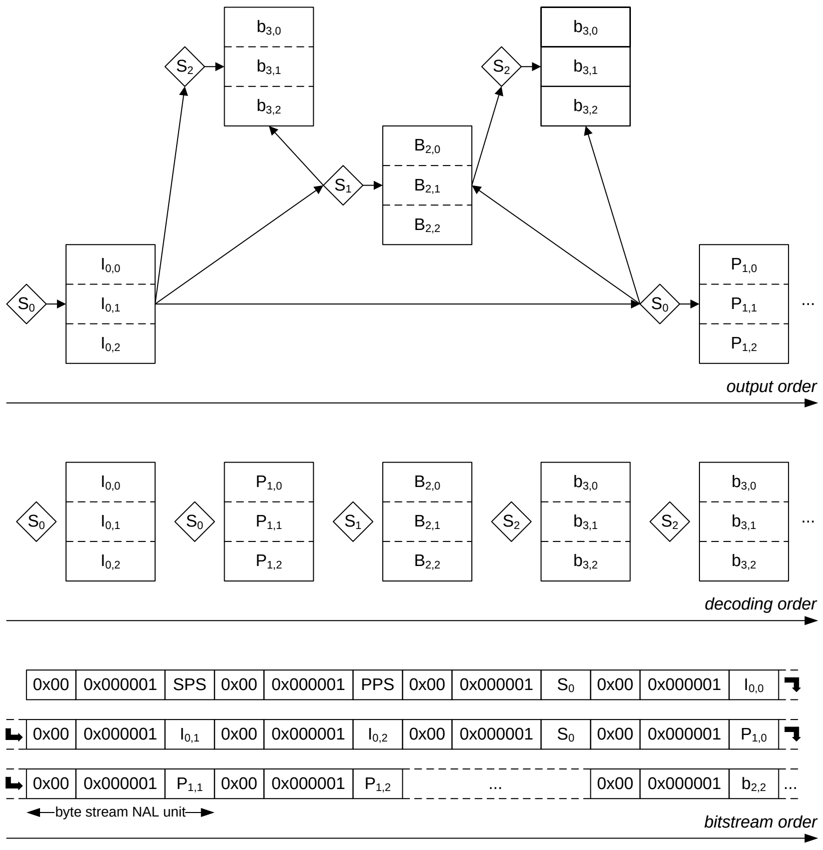


Figure 3.11: Use of sub-sequence information SEI messages for assigning a picture to a sub-sequence and a sub-sequence layer.

3.5.5 Summary

Thanks to the flexible design of the H.264/AVC video coding standard, several solutions are possible for the exploitation of temporal scalability, dependent on the targeted application and the features offered by a coded H.264/AVC bitstream. Similar observations can be made for other functionalities as well, such as the use of random access points in an H.264/AVC bitstream.

In this section, based on the information provided in the previous sections, we summarize six methods that can be used to exploit (multi-layered) temporal scalability in H.264/AVC. A clear insight in this adaptation process is fundamental for a good understanding of the use cases discussed further in this dissertation. The different temporal adaptation techniques will be mainly

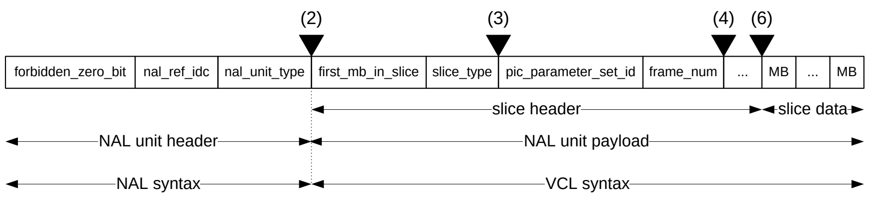


Figure 3.12: Level of detail needed by a number of temporal adaptation techniques.

discussed from the point of view of a bitstream parser, which may be part of a streaming server or which may be running on an intermediate network node. For a number of these techniques, the level of detail regarding the knowledge needed about the bitstream syntax is also visualized in Figure 3.12.

1. Bitstream switching using switching pictures is a first technique to adapt H.264/AVC bitstreams in the temporal domain. This approach does not imply the explicit removal of certain bitstream segments.

2. For the removal of non-reference pictures in an H.264/AVC bitstream, it is sufficient to parse a bitstream at the level of the NAL unit headers; the values of the `nal_ref_idc` and `nal_unit_type` syntax elements are used to identify the NAL units that belong to non-reference pictures.

3. For the elimination of non-reference B slice coded pictures, which is the traditional view of temporal scalability, it is necessary to parse up to and including the second syntax element of every slice header: the value of `slice_type` is required to identify the B slices in a bitstream; the value of `nal_ref_idc` is needed to verify whether a B slice is used as a reference or not; and the value of `nal_unit_type` is necessary to detect whether a NAL unit conveys a coded slice. This approach assumes that all slices of a picture share the same coding type. As previously discussed, the bitstream structure analysis can be simplified when access unit delimiters or SEI messages (see below) are employed in an H.264/AVC bitstream.

4. For discarding sub-sequence layers, implemented by using hierarchical B slice coded pictures but not signaled by using SEI messages, it is necessary to parse up to and including the fourth syntax element of every slice header; the values of `nal_ref_idc`, `nal_unit_type`, `slice_type`, and `frame_num` are necessary to determine to which layer the slices of a particular picture belong to.

5. For the SEI-driven disposal of sub-sequence layers, implemented by using a hierarchical coding pattern, it is sufficient to parse the bitstream at the level of the NAL unit headers. NAL units containing SEI messages have to be parsed completely. This can be considered the most elegant way to exploit multi-layered temporal scalability in H.264/AVC.

6. For the exploitation of temporal scalability using placeholder slices, it is necessary to completely parse every slice header (to know the bit boundary between the `slice_header()` and the `slice_data()` syntax structures). This technique is discussed in more detail in Chapter 6.

3.6 The scalable extensions to H.264/AVC

The JVT is currently in the process of finishing a new set of extensions to the H.264/AVC standard, called Scalable Video Coding (SVC)[13]. SVC addresses coding schemes for reliable delivery of video to diverse clients over heterogeneous networks, and in particular in scenarios where the downstream client capabilities, system resources, and network conditions are not known in advance. For example, clients may have different display resolutions, systems may have different caching or intermediate storage resources, and networks may have varying bandwidths and loss rates.

In the next two sections, for the sake of completeness, we give a brief overview of the design goals of the SVC amendment. We then proceed with giving an outline of its most important technical features. For more detailed information regarding SVC, we would like to refer the reader to [86], which is a document containing a draft of the SVC amendment, and to the Joint Scalable Video Model (JSVM) [87], which is a document containing background information regarding some of the algorithms and tools used in SVC[14]. It is expected that the SVC amendment will reach technical maturity around the beginning of 2007.

3.6.1 Design philosophy and technical features

The purpose of the SVC amendment is to define a limited number of coding tools on top of the H.264/AVC standard, offering several types of scalability at the level of a single bitstream [59]. The SVC design builds upon an

[13]In this dissertation, SVC is used to refer to the scalable extensions to H.264/AVC. However, SVC is sometimes used to refer to the more global concept of scalable video coding as well [85].

[14]In March 2007, a special issue of *IEEE Transactions on Circuits and Systems for Video Technology* will appear on SVC.

H.264/AVC-compatible base layer, and recycles existing building blocks such as motion compensation, transform coding, quantization, and entropy coding.

- The first version of the H.264/AVC standard already has extensive support for temporal scalability. For instance, hierarchical B pictures allow the straightforward creation of an H.264/AVC bitstream with multiple temporal enhancement layers. A high coding efficiency can be obtained thanks to the fact that most of the advanced prediction techniques of H.264/AVC can still be used to minimize the temporal correlation across the different enhancement layers. Therefore, SVC inherits the features for temporal adaptation that were already present in the first version of H.264/AVC. Only a limited number of additional syntax elements are introduced in the VCL to communicate the different temporal enhancement layers to a decoder or bitstream extractor (see below).

- Two different types of quality scalability are defined in the SVC amendment. The concept of scaling the visual content quality by omitting the transport and decoding of entire enhancement layers is denoted as Coarse-Grained quality Scalability (CGS). In some cases, the bit rate of a given enhancement layer can be reduced by truncating bits from individual NAL units. Truncation leads to a graceful degradation of the video quality of the reproduced enhancement layer. This concept is known as Fine-Grained quality Scalability (FGS).

- The first version of the H.264/AVC standard offers little support for spatial scalability: an adjustment of the spatial resolution of a bitstream is only possible by sending a new IDR picture and a new SPS (and eventually new PPSs) to switch to a different bitstream having another spatial resolution. Therefore, the SVC amendment incorporates a number of more advanced spatial scalability features, for instance allowing to reduce the spatial resolution of a bitstream from 4CIF (704×576) to CIF (352×288) using straightforward bitstream editing operations. Similar to the implementation of temporal scalability, cross-layer techniques are used to reduce the redundancy between different spatial enhancement layers, such as inter-layer prediction of residual data [31, 36]. Further, SVC also aims at offering support for features such as non-dyadic spatial scalability and cropping, which are denoted by the collective term Extended Spatial Scalability (ESS).

- FMO type 2 allows to make a distinction between different regions of importance in a sequence of pictures. This property can be exploited by encoders, which may assign a large bit budget to the regions of interest and a small bit budget to the regions of disinterest. At first sight, this coding tool may be employed to implement ROI scalability - this is, to enable the extraction of one or more ROIs from an H.264/AVC bitstream. However, the first version of the H.264/AVC specification does not allow any missing slice groups. This means that slice groups containing non-ROIs still need to be coded and to be present in an H.264/AVC bitstream in order not to violate the constraints of the standard. SVC relaxes this requirement and allows storing slice groups containing non-ROIs in enhancement layers. The base layer, containing the slices carrying the ROI data, remains compatible with the H.264/AVC standard.

- SVC includes combined scalability structures. These features make it possible to adapt a bitstream along multiple scalability axes at the same time. For example, a bitstream having features for spatio-temporal scalability can first be adapted in the temporal dimension, after which truncation operations can be applied along the spatial axis.

The new SVC amendment intends to achieve characteristics that are similar to the single-layer design of the H.264/AVC standard in terms of complexity, end-to-end delay characteristics, and robustness to transmission errors and congestion [5]. Also, a coding efficiency is to be achieved within about 10% excess bit rate for the same decoded video fidelity as non-scalable H.264/AVC. Meeting this objective will significantly minimize the coding efficiency gap compared to single-layer coding. Furthermore, when applicable, bitstream truncation is to be made possible at the level of the network abstraction layer.

3.6.2 Bitstream structure

In terms of coding structure, an SVC bitstream is composed of a base layer and at least one enhancement layer. The base layer typically represents the minimal temporal[15] and/or spatial resolution and/or quality of the SVC bitstream. This layer is independently decodable without the requirement of using any other layer of the SVC bitstream. Because the scalability extensions are built on H.264/AVC in a backwards-compatible way, a single-layer H.264/AVC decoder (as currently specified) is capable of decoding the base layer by ignoring the parts of the bitstream that correspond to the enhancement layer(s).

The parts of the SVC bitstream that belong to the enhancement layer(s) are conveyed by two new NAL unit types. A first new NAL unit type is defined to carry the coded slices of IDR pictures that are part of an enhancement layer, while a second new NAL unit type is introduced to transport the coded slices of

[15]Unless hierarchical coding patterns are used in the base layer.

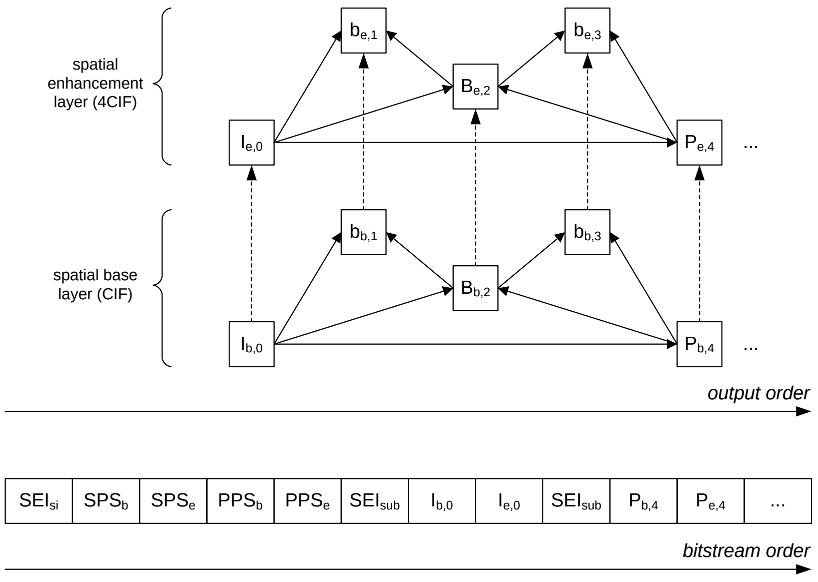


Figure 3.13: SVC bitstream (based on [21]): b (e) stands for base layer (enhancement layer). SEI$_{si}$ denotes a scalability info SEI message, while SEI$_{sub}$ represents a sub-sequence information SEI message. A full (dashed) arrow visualizes a temporal (spatial) dependency. The numerical picture indexes indicate the output order.

non-IDR pictures. As such, a bitstream extractor or decoder can easily determine whether a NAL unit belongs to the base layer or to an enhancement layer of an SVC bitstream by inspecting the NAL unit type. Besides two additional NAL unit types, the SVC amendment also introduces four new slice types: EI (I slice in scalable extension), EP (P slice in scalable extension), EB (B slice in scalable extension), and PR (progressive refinement slice).

In Figure 3.13, a coded video sequence is visualized containing two spatial layers. The spatial base layer, compliant with single-layered H.264/AVC, contains two temporal enhancement layers (i.e., three sub-sequence layers). The corresponding bitstream structure is provided as well, which is, similar to a bitstream compliant with the first version of the H.264/AVC standard, a sequence of NAL units separated from each other using zero-valued bytes and a start code. Three categories of NAL units can be distinguished [21].

1. The first category contains NAL units carrying SPSs and PPSs. Among other parameters, an SPS conveys the spatial resolution, which is different for each spatial enhancement layer. Therefore, the bitstream is visualized in Figure 3.13 contains two SPSs, one for each spatial layer.

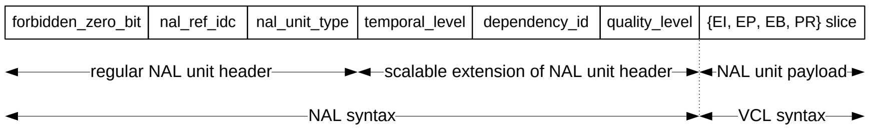


Figure 3.14: Scalable extension of the regular NAL unit header.

2. The second category contains NAL units having coded video data as payload. These NAL units can, in their turn, also be classified into two sets: on the one hand, NAL units containing coded slices that are part of the H.264/AVC-compliant base layer, and on the other hand, NAL units containing coded slices that belong to an enhancement layer.

   In contrast to the slices that are part of the H.264/AVC-compliant base layer, the slices that are residing in an enhancement layer carry information regarding the layer they are embedded in. This information is stored in a number of syntax elements, which are part of an extension of the regular NAL unit header (shown in Figure 3.14). This extension is only defined for the two newly introduced NAL unit types. The values of the `temporal_level`, `dependency_id`, and `quality_level` syntax elements assist a bitstream extractor in generating a bitstream that is tailored for a particular usage environment.

3. The third category contains NAL units conveying SEI messages. A number of these SEI messages can be used by a bitstream extractor for the efficient adaptation of an SVC bitstream. For instance, in Figure 3.13, the first NAL unit in the bitstream contains a scalability info SEI message (see below); all other SEI messages in the SVC bitstream are sub-sequence information SEI messages.

A bitstream extractor is a tool that enables the creation of a bitstream of lower bit rate and lower quality, and possibly with lower temporal or spatial resolution from the input bitstream according to the desired extraction option, e.g. a given target bit rate at a given spatial resolution and a given picture rate. To prevent a bitstream extractor from having to do a complex analysis of the bitstream structure, an SVC encoder will typically provide a scalability information SEI message at the start of an SVC bitstream. In a manner, the scalability information SEI message can be seen as an extended version of the sub-sequence layer information SEI message, as previously discussed.

By analyzing the statistical information contained in the scalability information SEI message, a single-loop bitstream extractor can be created that is able to acquire an in-depth overview of the structure of the scalable bitstream, including information pertaining to the different scalability features offered by the bitstream. Consequently, a scalability information SEI message simplifies the extraction process that is required to create a target bitstream suited for playback in a particular usage environment.

A NAL unit that contains a slice that is part of the base layer does not convey any information regarding the temporal layer it belongs to. This is due to the fact that the syntax element `temporal_level` is missing in the NAL unit syntax for the base layer. Therefore, a sub-sequence information SEI message is used as an alternative to the `temporal_level` syntax element to communicate the sub-sequence layer organization to a bitstream extractor or decoder. In Figure 3.13, sub-sequence information SEI messages are employed to assign pictures of the H.264/AVC-compliant base layer to one of the three temporal enhancement layers available within the spatial base layer.

The bitstream extraction process in SVC can essentially be summarized in the three steps described below.

- First, the bitstream extractor receives information about the constraints of the targeted usage environment (e.g., in terms of desired bit rate).
- The bitstream extractor subsequently analyzes the scalability information SEI message. Using the information regarding the constraints of the usage environment, appropriate values are determined for the `temporal_level`, `dependency_id`, and `quality_level`.
- Finally, the bitstream extractor processes the remaining NAL units of the scalable video bitstream. NAL units with feasible values for `temporal_level`, `dependency_id`, and `quality_level` are kept (compared to the values for `temporal_level`, `dependency_id`, and `quality_level` as derived during the analysis of the scalability information SEI message).

For a more thorough discussion regarding the bitstream extraction process in SVC, we would like to refer the interested reader to [21].

3.7 Conclusions and original contributions

Support for true bitstream scalability, which is functionality to achieve temporal, spatial, or SNR scalability by simply discarding parts of a coded bitstream, was not incorporated in the first version of the H.264/AVC standard, due to the strict time schedule of the standardization process. Nonetheless, a number of adaptivity mechanisms are incorporated in the initial H.264/AVC specification. Four of these adaptivity provisions, which are mainly targeting bit rate adaptation, were reviewed in this chapter:

- switching pictures for bitstream switching;
- FMO for ROI coding (and extraction; see Chapter 6);
- data partitioning for offering coarse-granularity quality scalability;
- sub-sequence layers for achieving multi-layered temporal scalability.

During this review, we reported the key results of a number of experiments, illustrating the flexibility of use of FMO for the purpose of error resilience and content adaptation. Further, we also demonstrated that data partitioning can be used as a coarse but simple and efficient technique for achieving quality scalability. Particular attention was also paid to the implementation of multi-layered temporal scalability in H.264/AVC bitstreams, using sub-sequences and sub-sequence layers, coding patterns based on hierarchical B pictures, and SEI messages. This discussion can be considered the main contribution of this chapter: it provides a *complete* and *coherent* overview on how to employ temporal scalability in H.264/AVC bitstreams.

The importance of sub-sequences and sub-sequence layers lies in the context of video coding formats that allow the use of sophisticated prediction strategies for the purpose of reducing the temporal correlation. The resulting temporal dependencies significantly increase the complexity of the decision process regarding the disposal of certain pictures or picture chains. In H.264/AVC, complex prediction structures are often implemented using hierarchical B pictures, although more arbitrary prediction patterns are possible as well. The concept of sub-sequences and sub-sequence layers, which are units of content adaptation, is particularly useful in this context: they enable the easy identification and disposal of chains of pictures, while still allowing the use of efficient prediction techniques. As such, sub-sequences and sub-sequence layers assist in creating a balanced trade-off between coding efficiency on the one hand and bit rate adaptation using temporal scalability on the other hand.

A number of SEI messages are used to communicate the bitstream organization, in terms of sub-sequences and sub-sequence layers, to a bitstream extractor or decoder. These content adaptation hints assist in abstracting the complexity of the underlying bitstream format: they allow to think about a bitstream in terms of its structure (e.g., headers, packets, and layers of data), and not in terms of coding concepts (e.g., entropy coding, transform coefficients, motion vector differences, and so on).

Furthermore, we identified six methods that can be used to exploit (multi-layered) temporal scalability in H.264/AVC bitstreams. A clear insight in these temporal adaptation techniques is fundamental for a good understanding of a number of use cases discussed further in this dissertation.

Finally, in this chapter, we also outlined the most important design principles and features of the scalable extensions to H.264/AVC. These extensions define a limited number of coding tools on top of the initial H.264/AVC standard, adding support for features such as spatial and fidelity scalability. Temporal adaptivity provisions are inherited from the first edition of the H.264/AVC specification. Bitstream adaptation in SVC is achieved using simple editing operations at the level of the bitstream, a process that is again guided by relying on SEI metadata messages.

In the next chapters, the principles of Bitstream Syntax Description-based content adaptation will be introduced and extensively discussed. Existing techniques, as well as own ideas, will be evaluated by relying on different use cases. As previously mentioned, one of the most important use cases involves the description-driven exploitation of multi-layered temporal scalability in H.264/AVC, using hierarchical B pictures and SEI messages. We will also show that there is a clear connection between the design philosophy behind SEI messages and the idea of description-based content adaptation.

Our research in this domain resulted in contributions that are incorporated in the publications listed below. A number of papers are deliberately omitted as they are enumerated in one of the subsequent chapters.

1. Stefaan Mys, Peter Lambert, Wesley De Neve, Piet Verhoeve, Rik Van de Walle. SNR Scalability in H.264/AVC Using Data Partitioning. In *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, Volume 4261, pages 333–343, October 2006.

2. Koen De Wolf, Davy De Schrijver, Wesley De Neve, Rik Van de Walle. Adaptive Residual Interpolation: A Tool For Efficient Spatial Scalability In Digital Video Coding. In *Proceedings of The 2006 International Conference on Image Processing, Computer Vision, & Pattern Recognition (IPCV'06)*, pages 131–137, Las Vegas, Nevada, USA, June 2006.

3. Peter Lambert, Wesley De Neve, Yves Dhondt, Rik Van de Walle. Flexible Macroblock Ordering in H.264/AVC. *Journal of Visual Communication & Image Representation*, 17:358-375, January 2006.

4. Koen De Wolf, Robbie De Sutter, Wesley De Neve, Rik Van de Walle. Comparison of Prediction Schemes with Motion Information Reuse for

Low Complexity Spatial Scalability. In *Proceedings of SPIE/Visual Communications and Image Processing*, Volume 5960, pages 1911–1920, Beijing, China, July 2005.

Chapter 4

BSD-driven media resource adaptation

Adapt or perish, now as ever, is nature's inexorable imperative.

Herbert G. Wells (1866-1946), Mind at the End of Its Tether, 1945.

4.1 Introduction

Scalable coding formats have always been a major point of interest in the world of digital media content. Scalability features are available in several standards for still image coding (e.g., JPEG 2000 [88]), audio coding (e.g., MPEG-4 Bit Sliced Arithmetic Coding, abbreviated as MPEG-4 BSAC [89]), and video coding (e.g., H.262/MPEG-2 Video, H.263+, MPEG-4 Visual). A scalable bitstream is defined in MPEG-21 DIA as: "*A bitstream in which data is organized in such a way that, by retrieving the bitstream, it is possible to first render a degraded version of the resource, and then progressively improve it by loading additional data*".

Scalable bitstreams are intended to pave the way for the deployment of several new multimedia architectures. These architectures will make it possible to tackle the tremendous diversity in terminals and networks used in present-day and future multimedia ecosystems. However, an efficient solution for dealing with this heterogeneity does not only imply the use of scalable coding formats, but it also requires the employment of an Adaptation Decision Taking Engine (ADTE; [90]) and a content adaptation system [91]. These complementary technologies aim at formulating an answer to the question on how to optimally and efficiently adapt a scalable bitstream to a given set of constraints (e.g., device and network capabilities, user characteristics and preferences, and natural environment characteristics) [92].

The adaptation of scalable bitstreams usually involves a number of straightforward editing operations. These operations typically consist of the removal of certain data segments and the modification of the values of particular syntax elements, without requiring the compressed media data to be recoded. This process can for instance be realized by relying on automatically generated textual descriptions that contain information about the high-level syntax of a scalable bitstream [93]. These structural metadata, typically expressed using XML, can subsequently be transformed to reflect a desired adaptation of a scalable bitstream, and can then be used to automatically create an adapted version of the bitstream. As such, the textual descriptions, further referred to as Bitstream Syntax Descriptions (BSDs), act as an intermediate layer for adapting (scalable) bitstreams to the constraints imposed by a particular usage environment (see Figure 4.1).

In this chapter, we first describe the principles of BSD-driven media resource adaptation. Next, two different approaches are outlined for BSD-based content customization: a framework using the standardized MPEG-21 Bitstream Syntax Description Language (MPEG-21 BSDL) and a framework relying on the Formal Language for Audio-Visual Object Representation, extended with XML features (commonly abbreviated as XFlavor). MPEG-21 BSDL is explained in detail because of the activities of the Multimedia Lab research group, which are closely aligned to MPEG standardization. As such, MPEG-21 BSDL was used as the starting point for our study of the principles of BSD-driven content adaptation. In a next step, we discuss a number of original contributions in the domain of BSD-based media resource adaptation, in particular the use of MPEG-21 BSDL in the context of several state-of-the-art media formats (file formats and video coding formats). Before concluding this chapter, a performance analysis is provided regarding an implementation of the MPEG-21 BSDL and XFlavor tool chains for the BSD-driven exploitation of temporal scalability in Video Codec 1 (VC-1; [94]), a new video coding format that was standardized in 2006 by the Society of Motion Pictures and Television Engineers (SMPTE).

Finally, for a good understanding of this chapter, we assume some awareness of the basic design principles of a few popular MPEG media formats (e.g., H.262/MPEG-2 Video [2], MPEG-2 Systems [95]), as well as of a number of XML technologies such as W3C XML Schema [96], XML Path language (XPath; [97]), and eXtensible Stylesheet Language Transformations (XSLT; [98]).

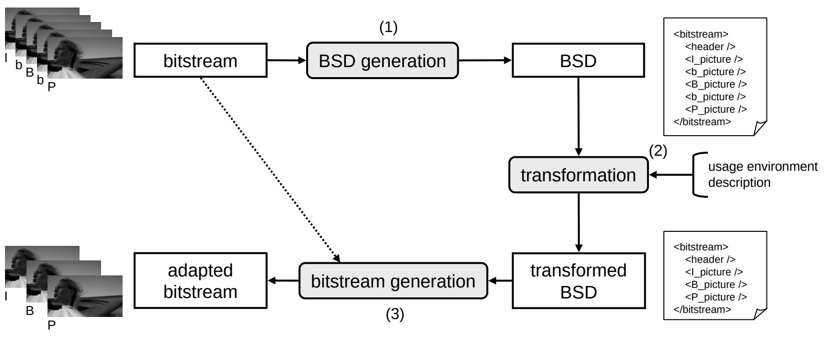


Figure 4.1: Generic architecture for a BSD-based content adaptation system, bridging the gap between content (media resources) and context (usage environment).

4.2 Principles of BSD-based content adaptation

In general, BSD generation, BSD transformation, and bitstream reconstruction are performed as shown in Figure 4.1. Explanatory notes are provided below:

1. automatic generation of a BSD, which closely reflects the bitstream structure (e.g., described in terms of layers, headers, or pictures);

2. BSD transformation, resulting in the disposal or modification of particular BSD fragments (e.g., removal of BSD fragments describing non-reference B pictures) to take into account the constraints of a particular usage environment;

3. adapted bitstream generation (e.g., creation of a coded video bitstream without non-reference B pictures), optionally using the original bitstream (denoted by a dashed arrow in the figure).

In the remainder of this chapter, Figure 4.1 will be further refined to describe the different architectures for BSD-driven media resource adaptation. Note that, for reasons of simplicity, the fictive BSDs, as used throughout the different figures, describe the coded bitstream structure in output order, and not in bitstream order.

The main advantages of using XML-based BSDs for media resource adaptation can be summarized as follows:

1. A BSD acts as an abstraction layer. This can be considered from two different points of view. First, a BSD abstracts the complexity of the

composition of the coded bitstream; its typical high-level nature only requires a limited knowledge regarding the bitstream structure. As such, it is for example not necessary to reason about a video bitstream in terms of transform coefficients or motion vectors, but it is possible to think about the bitstream as how it is organized in terms of packets, headers, or layers of data[1]. Second, a BSD abstracts the bitstream parsing process. As BSDs are typically generated in an automatic way, it is no longer required to implement cumbersome bitstream parsing operations in a particular programming language in order to discover the bitstream structure.

2. The complexity of the content adaptation step is shifted from the compressed domain to the XML domain. This enables the use of many already existing XML tools for manipulating BSDs, such as editors and transformation engines. It also allows to achieve a straightforward integration with other XML-oriented metadata standards, such as the MPEG-7 specification [99].

3. A format-neutral content adaptation engine can be implemented, which is a combination of a BSD transformation engine and a BSD-driven bitstream extractor. This format-independent property, which is the main advantage of a BSD-based content adaptation approach, implies that the same software modules can be used, regardless of whether adaptation is employed in the context of a scalable still image format, a scalable audio format, or a scalable video format.

   This also means that the program code of these software components does not have to be updated in order to support a new media format[2], which makes these modules particularly suited for an implementation in hardware.

The following reasons justify the use of XML for the representation of the structural metadata [100]:

- XML offers a flexible way for describing highly structured data.

[1]This objective is similar to the purpose of the different sub-sequence SEI messages in H.264/AVC, as well as to the purpose of the scalability information SEI message in SVC.

[2]The reuse of program code is an issue when transcoding is used as an alternative to the use of scalable coding. Transcoding routines are often application-specific and format-dependent, and their implementation is typically to be considered an awkward and error-prone task. However, transcoding is for example useful in an application scenario in which end-users are able to upload home-made video streams, using different video coding formats, to a content site, which relies on a single (scalable) video coding format for the purpose of efficient distribution.

- XML is the de facto language for describing metadata [101]. This way, BSDs can be easily integrated with already existing metadata standards. For instance, descriptive metadata in line with MPEG-7 can be used to implement semantic adaptations (e.g., removal of violent scenes).
- An extensive tool set is available for processing XML documents in a fast, reliable, and flexible way.
- XML is extensible: it is possible to add information to an existing XML document in such a way that applications that are not familiar with this new information will ignore it.
- XML is platform-independent as it is represented by means of plain text.

In addition to the exploitation of scalability, BSD-driven content adaptation also enables other applications. For instance, this approach can be used for correcting wrongly coded syntax elements without a need for reencoding. Think for example about the adjustment of aspect ratio information, about correcting four-character codes in file containers (FourCCs), and so on. These operations are sometimes referred to as header hacks.

Further possible applications with regard to BSD-based adaptation, but not restricting to, are format-independent streaming/hinting [102], multiplexing and demultiplexing, automatic video summarization, scene selection, bitstream syntax validation, and metadata injection. The last technique can for example be used to insert user data in MPEG-4 Visual elementary bitstreams or to add SEI messages to H.263+ or H.264/AVC bitstreams (see Chapter 5).

4.3 Bitstream syntax description languages

In recent years, a number of bitstream syntax description languages have been defined that enable XML-driven manipulation of digital media resources, such as XFlavor [103], the MPEG Video Markup Language (MPML; [104]), MPEG-21 BSDL, and MPEG-21 generic Bitstream Syntax Schema (MPEG-21 gBS Schema; [8]).

MPML is an XML application specifically designed for describing the syntax of bitstreams compliant with MPEG-4 Visual [4]. This language will not be discussed in further detail due to its rather specific nature. The main principles of BSDL and XFlavor are explained in the following sections. gBS Schema will be described briefly in the section pertaining to BSDL. Detailed information about BSDL and gBS Schema, and about the MPEG-21 Multimedia Framework in general, can be found in [105] and in a special issue of *IEEE Transactions on Multimedia* [106].

To allow the reader to do a side-by-side comparison of the language features of BSDL and XFlavor, a number of syntax fragments are listed in Appendix A. A description in BSDL of several important syntax structures of H.264/AVC can be found in Listing A.1. Listing A.2 contains a description of the same syntax structures in XFlavor. These descriptions are respectively used by a BSDL and XFlavor tool chain to automatically create BSDs for H.264/AVC bitstreams. The BSDs in question are depicted in Listing A.5 for BSDL and in Listing A.6 for XFlavor.

4.3.1 MPEG-21 BSDL

Introduction

MPEG-21 BSDL provides means for translating the high-level syntax of a binary media resource into an XML document [107]. The language is built on top of the World Wide Web Consortium's (W3C) XML Schema language and falls under the umbrella of the Bitstream Syntax Description tool of the MPEG-21 DIA specification. This standard envisions the construction of a generic content adaptation framework that is entirely steered by XML technologies. The goal of DIA is in its turn in line with the vision of the MPEG-21 Multimedia Framework, which aims at enabling a transparent and augmented use of multimedia resources across a wide range of networks and devices, by means of the creation of a multimedia framework that consists of generic software components, communicating with each other using XML-based messages that capture the format-dependent specificities.

The primary motivation behind the development of BSDL is to assist in the adaptation of scalable bitstreams, such that the resulting bitstreams meet the constraints imposed by a particular usage environment. The generic character of MPEG-21 BSDL, and hence its merit, lies in the format-independent nature of the logic that is responsible for the creation of BSDs and for the generation of the adapted bitstreams. More precisely, these different pieces of logic do not need to be updated or modified in order to support a new (scalable) media format in a BSDL-based content adaptation framework. This is due to the fact that all information, necessary for discovering the structure of a bitstream, is available in a document called a Bitstream Syntax Schema (BS Schema).

A BS Schema contains a description of (a part of) the syntax of a particular media format in the MPEG-21 BSDL schema language. It conveys, among other things, information about the name-giving of syntactical structures, their type and binary encoding, and their position in the media format hierarchy. As such, the MPEG-21 BSDL specification enables the construction of a format-independent processor to automatically parse a bitstream and to generate its

XML description. It also allows the creation of a processor that is unaware of a specific media format in order to generate an adapted bitstream from its transformed XML description. The BSD and bitstream generator are named BintoBSD Parser and BSDtoBin Parser, respectively.

Listing 4.1: BS Schema fragment for a sub-sequence information SEI message.

```
<xsd:element name="sub_seq_info">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="sub_seq_layer_num"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="sub_seq_id"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="first_ref_pic_flag" type="b1"/>
      <xsd:element name="leading_non_ref_pic_flag" type="b1"/>
      <xsd:element name="last_pic_flag" type="b1"/>
      <xsd:element name="sub_seq_frame_num_flag" type="b1"/>
      <xsd:element name="sub_seq_frame_num_flag" minOccurs="0"
           bs2:if="./jvt:sub_seq_frame_num_flag = 1"
           type="jvt:UnsignedExpGolomb"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

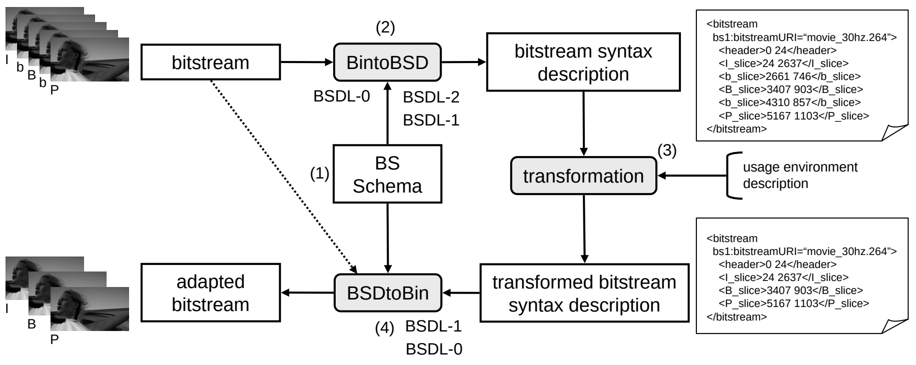
Listing 4.2: BSD fragment for a sub-sequence information SEI message.

```
<sub_seq_info>
  <sub_seq_layer_num>0</sub_seq_layer_num>
  <sub_seq_id>0</sub_seq_id>
  <first_ref_pic_flag>0</first_ref_pic_flag>
  <leading_non_ref_pic_flag>0</leading_non_ref_pic_flag>
  <last_pic_flag>0</last_pic_flag>
  <sub_seq_frame_num_flag>0</sub_seq_frame_num_flag>
</sub_seq_info>
```

A BS Schema fragment is provided in Listing 4.1; it describes the formal syntax of a sub-sequence information SEI message[3]. Listing 4.2 contains a corresponding BSD excerpt, obtained after having parsed a sub-sequence information SEI message in an H.264/AVC bitstream using a BintoBSD Parser.

Figure 4.2 summarizes the overall method for adapting a scalable bitstream using MPEG-21 BSDL, illustrating the removal of particular bidirectionally

[3]For comparison, the tabular syntax of a sub-sequence information SEI message, as used in the H.264/AVC standards document, is provided in Table 3.3 in Chapter 3.



```
<bitstream
  bs1:bitstreamURI="movie_30hz.264">
    <header>0 24</header>
    <I_slice>24 2637</I_slice>
    <b_slice>2661 746</b_slice>
    <B_slice>3407 903</B_slice>
    <b_slice>4310 857</b_slice>
    <P_slice>5167 1103</P_slice>
</bitstream>
```

```
<bitstream
  bs1:bitstreamURI="movie_30hz.264">
    <header>0 24</header>
    <I_slice>24 2637</I_slice>
    <B_slice>3407 903</B_slice>
    <P_slice>5167 1103</P_slice>
</bitstream>
```

Figure 4.2: BSD-driven content (i.e., video) adaptation with BSDL. Note how the BSDs refer to data segments of the original input bitstream.

coded pictures to create a tailored video bitstream suited for playback in a constrained usage environment. Explanatory notes are provided below:

1. an MPEG-21 BS Schema contains a description of the high-level syntax of a particular media format;

2. a BSD is created by a format-independent BintoBSD Parser, taking as input a particular bitstream and a corresponding BS Schema;

3. a BSD is transformed to meet the constraints of a certain usage environment (the way the BSD is transformed is not standardized by DIA);

4. a format-independent BSDtoBin Parser is used for creating an adapted bitstream, using the transformed BSD and the BS Schema, optionally taking the original bitstream as an additional input (denoted by the dashed arrow).

MPEG-21 BSDL only addresses the description of the high-level structure of a bitstream; the schema language is typically used to describe a restricted subset of the syntax of a particular media format, the exact level of detail dependent on the application targeted. This means that the description itself is scalable (i.e., its granularity can be adjusted), making it possible to avoid a large overhead and the execution of unnecessary computations. Consequently, a BSD created in the context of MPEG-21 BSDL acts as an additional metadata layer that is not meant to replace the original bitstream.

MPEG-21 DIA also contains a second tool for describing the syntax of a media resource - MPEG-21 gBS Schema. This language allows the creation

Table 4.1: Relevant namespaces.

specification	namespace	prefix
XML Schema	http://www.w3.org/2001/XMLSchema	`xsd`
BSDL-0	urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS	`bs0`
BSDL-1	urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS	`bs1`
BSDL-2	urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS	`bs2`

of format-independent BSDs, so-called generic BSDs (gBSDs). Individual elements in the gBSDs can also be marked to enable or simplify semantic-based and complex adaptations. However, gBS Schema does not provide a BS Schema that would allow the BintoBSD operation to be performed on a bitstream. gBSDs must be created by some other (unspecified) means (e.g., using an application-specific parser; see [93] for an in-depth overview).

The principles of MPEG-21 BSDL have been illustrated in [107] for the adaptation of JPEG 2000 still images and MPEG-4 Visual compliant bitstreams. An example application of BSDL to MPEG-4 Visual Texture Coding (MPEG-4 VTC) was introduced in [108], and an application of BSDL to the experimental wavelet-based Motion Compensated Embedded Zero Block Coding (MC-EZBC) codec was discussed in [16]. A harmonized BSDL/gBS Schema approach was outlined in [93], while [109] briefly discusses the use of BSDL for the adaptation of audio bitstreams compliant to MPEG-4 BSAC. These examples illustrate the expressive power of the MPEG-21 BSDL framework, enabling BSD-driven content adaptation. Regardless of the bitstream format used, the same format-agnostic media processors can be deployed for BSD generation and bitstream adaptation.

Language specification

MPEG-21 BSDL is built by introducing two normative successive sets of extensions and restrictions over the W3C XML Schema language. The extensions are expressed by means of two XML Schemas, while the restrictions are fixed in the standards document itself. The *Schema for BSDL-1 Extensions* defines a number of attributes and data types that can be used in a BS Schema, but that do not exist in XML Schema. The BSDtoBin process needs these extensions for the creation of correct bitstreams from a (transformed) BSD. The *Schema for BSDL-2 Extensions* defines a number of additional extensions to XML Schema that are needed to resolve ambiguities in the BintoBSD process. For the sake of completeness, a number of relevant namespaces and corresponding prefixes can be found in Table 4.1.

Table 4.2: Overview of all XML Schema data types allowed in BSDL.

data type	length (bytes)
`xsd:string`	unlimited
`xsd:float`	4
`xsd:double`	8
`xsd:hexBinary`	unlimited
`xsd:base64Binary`	unlimited
`xsd:long`	8
`xsd:int`	4
`xsd:short`	2
`xsd:byte`	1
`xsd:unsignedLong`	8
`xsd:unsignedInt`	4
`xsd:unsignedShort`	2
`xsd:unsignedByte`	1

Restrictions on XML Schema Some constructs and data types that may occur in an XML document and that can be expressed in an XML Schema are not useful for BSDs as they may cause certain ambiguities. Therefore, BSDL defines a number of restrictions on the XML Schema language.

A first restriction is that bitstream data have to be embedded in a BSD as element content, and not in an attribute. The reason for this is that the order of attributes in an XML document is not significant. As such, an external knowledge would be required to specify in what order attributes should be processed in case they would contain symbols to be added to the output bitstream.

Mixed content is not allowed either, as all elements must be assigned a type. Therefore, `xsd:mixed` cannot occur in a BS Schema. Elements with no type, expressed by means of `xsd:any`, `xsd:anyType`, or `xsd:anySimpleType` are not allowed either.

A limited number of data types that exist in XML Schema are allowed in BSDL. Only those for which a binary representation is possible, are permitted. For instance, `xsd:integer` is not allowed, as the number of bytes needed for representing such an element is not specified. An element of type `xsd:int` is allowed, because in this case, the number of bytes is fixed. In Table 4.2, all data types that belong to XML Schema and that are allowed to occur in a BS Schema, are shown. Some of these data types do not have an a priori length. If such a data type occurs in a BS Schema, the BintoBSD process will need information belonging to the BSDL-2 Extensions for determining the actual length. This problem does not occur for BSDtoBin as the length of the data stream becomes clear when evaluating the element content.

Listing 4.3: Example of the use of `xsd:maxExclusive` in BSDL.

```
<!-- The following data type consists of 18 bits. -->
<xsd:simpleType name="b18">
  <xsd:restriction base="xsd:unsignedInt">
    <xsd:maxExclusive value="262144"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- Use of the b18 data type. -->
<xsd:element name="time_code" type="m4v:b18"/>
```

In addition, other data types can be defined in a BS Schema, by restricting the range of one of the existing unsigned data types, using the `xsd:maxExclusive` restriction mechanism of XML Schema. Listing 4.3 shows how to define a new data type that consists of exactly 18 bits, needed for the representation of the `time_code` syntax element in the `group_of_video_object_plane` syntax structure of MPEG-4 Visual.

BSDL-1 The Schema for BSDL-1 Extensions defines two new data types: `bs1:byteRange` and `bs1:bitstreamSegment`, both used for referring to fragments of the original bitstream. Elements of the `bs1:byteRange` data type consist of two non-negative integer values. The first value refers to a location in the current bitstream, represented as a byte offset, where the fragment begins, and the second value indicates the length, in bytes, of this fragment. A `bs1:bitstreamSegment` element has the same functionality, but elements of this data type have a `bs1:start` and a `bs1:length` attribute. This data type only exists for compatibility with gBS Schema, which imports the Schema for BSDL-1.

The location of the bitstream, referred to by `bs1:byteRange` or `bs1:bitstreamSegment`, is signaled by means of the `bs1:bitstreamURI` attribute. This attribute may occur in the root element of a BSD, thus providing a default value for that particular description, but it may also occur in a `bs1:byteRange` or a `bs1:bitstreamSegment` construct, thus overwriting the default location of the bitstream referred to.

A final extension defined in the Schema for BSDL-1 Extensions is the `bs1:ignore` attribute. Elements in a BSD that carry this attribute will be ignored by the BSDtoBin process if its value is set to `true`, either in the BSD itself, or in the BS Schema by setting its default value to `true`. This way, it is possible to add metadata information to a BSD in such a way that it does not influence the generation of the (adapted) bitstream itself.

BSDL-2 The most important extension in BSDL-2 is the introduction of support for XPath, a language for addressing parts of an XML document. Sometimes, the presence of a particular syntax element in a bitstream, its number of occurrences, or its length, may be dependent on the value of another syntax element. Such cases can be expressed in a BS Schema using XPath expressions. BintoBSD needs these expressions for generating correct BSDs.

The `bs2:nOccurs` attribute is a first attribute that takes an XPath expression as value. It is used when the number of occurrences of a particular element depends on the value of another element. If it is absent, its value is supposed to be one. In order to make sure that the BS Schema is a correct XML Schema for the produced BSDs, the values for `xsd:minOccurs` and `xsd:maxOccurs` must correspond with all possible values for `bs2:nOccurs`.

Listing 4.4: Example of the use of `bs2:ifNext` in BSDL.

```
<!-- Declaration of byte_stream_nal_unit. -->
<xsd:element name="byte_stream_nal_unit">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="zero_byte" type="xsd:unsignedByte"
           fixed="0" minOccurs="0" maxOccurs="unbounded"
           bs2:ifNext="000000"/>
      <xsd:element name="start_code_prefix_one_3bytes"
           type="jvt:StartCodeType" fixed="000001"
           minOccurs="0" maxOccurs="1"
           bs2:ifNext="000001"/>
      <xsd:element ref="jvt:nal_unit"
           minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Often, elements of a bitstream are conditional. In a BS Schema, this is expressed by means of the `bs2:if` or the `bs2:ifNext` attributes. An element containing `bs2:if` only occurs in a BSD if the XPath expression in the attribute evaluates to `true`. A `bs2:ifNext` attribute contains a hexadecimal value or a range of hexadecimal values (two values, separated by white space), as shown in Listing 4.4 for the parsing of the `byte_stream_nal_unit` syntax structure in H.264/AVC. In this case, the element will only occur if the following bytes in the bitstream correspond with the value or range of values mentioned in the `bs2:ifNext` attribute. Similar to `bs2:nOccurs`, `xsd:minOccurs` and `xsd:maxOccurs` ought to have appropriate values, if the BS Schema is to be employed as a valid XML Schema.

Another attribute defined in the Schema for BSDL-2 Extensions is the `bs2:rootElement` attribute in the `xsd:schema` element. This attribute tells the BintoBSD process which element of the schema should be used as the root element of the BSD.

In XML Schema, a number of data types can have an unlimited representation size, as can be seen in Table 4.2. The same is true for the `bs1:byteRange` data type. In these cases, the BintoBSD process needs additional information pertaining to the length of the actual syntax elements, otherwise all remaining bytes in the bitstream are considered to be part of that element. Because XML Schema does not support such constructs directly, the use of `xsd:annotation/xsd:appinfo` inside an `xsd:restriction` element is needed.

One of the mechanisms that can be used for limiting the range of unlimited data types is the `bs2:length` attribute, which may contain an XPath expression for determining the number of bytes the element consists of. In Listing 4.5, the use of the `bs2:length` attribute is shown for determining the length of a `bs1:byteRange` data type: an XPath expression is used to compute the length of a Packetized Elementary Stream packet (PES packet) in a private stream, which is in its turn part of an MPEG-2 Program Stream.

Listing 4.5: Example of the use of `bs2:length` in BSDL.

```
<!-- Declaration of private_stream_1. -->
<xsd:element name="private_stream_1">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="packet_start_code_prefix"
           type="mp2:StartCodePrefixType"/>
      <xsd:element name="stream_id"
                   type="mp2:StartCodeSuffixType"/>
      <xsd:element name="PES_packet_length" type="b16"/>
      <xsd:element name="PES_packet_payload">
        <xsd:simpleType>
          <xsd:restriction base="bs1:byteRange">
            <xsd:annotation>
              <xsd:appinfo>
                <bs2:length value="../mp2:PES_packet_length"/>
              </xsd:appinfo>
            </xsd:annotation>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Another way of determining the length of an unlimited data type is the use of the `bs2:startCode` or `bs2:endCode` language features of BSDL. When the length of an element is restricted using these constructs, the BintoBSD process looks for the next occurrence of the hexadecimal pattern or range that is mentioned in the `bs2:value` attribute of the `bs2:startCode` or `bs2:endCode` element. This type of parsing is called delimiter-driven parsing, which is the opposite of parsing based on the use of length fields. When `bs2:startCode` is used, the start code pattern does not belong to the current element; in case of a `bs2:endCode`, it does. An example of the use of start codes in BSDL is given in Listing 4.6, used for identifying the beginning of a picture or a slice in MPEG-1 Video. The end of the payloads is determined by a number of start code values, which can either fall within a given range (from 0x00000101 up to 0x000001AF) or correspond with a fixed value (e.g., 0x00000100). Note that the `bs2:length` attribute, as discussed in the previous paragraph, cannot be used in combination with the `xsd:length`, `bs2:startCode`, or `bs2:endCode` language constructs.

Listing 4.6: Example of the use of `bs2:startCode` in BSDL.

```
<!-- Declaration of PictureAndSlicePayloadType. -->
<xsd:simpleType name="PictureAndSlicePayloadType">
  <xsd:restriction base="bs1:byteRange">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:startCode value="00000100"/>
        <bs2:startCode value="00000101 000001AF"/>
        <bs2:startCode value="000001B8"/>
        <bs2:startCode value="000001B3"/>
        <bs2:startCode value="000001B7"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>
<!-- Use of the PictureAndSlicePayloadType data type. -->
<xsd:element name="slice_payload"
     type="mp1:PictureAndSlicePayloadType"/>
```

A final construct that belongs to BSDL-2 is `bs2:ifUnion`. This language feature is similar to the well-known `switch/case` construct in most programming languages, and is used when an element of a BS Schema is defined using `xsd:union`. In that case, one type from a list of types, enumerated in the `bs2:memberTypes` attribute, is allowed to occur. In order to be able to select the appropriate type, BintoBSD needs the information that occurs in the `bs2:ifUnion` elements. The index of the first `bs2:ifUnion` element for which the XPath expression evaluates to `true` will determine which element will be selected. If all n expressions evaluate to false, the member type with position $n + 1$ is selected. This way, this element is actually treated as a default type.

Listing 4.7: Example of the use of `bs2:ifUnion` in BSDL, using simplified XPath expressions ($SPS refers to the active SPS).

```
<!-- Declaration of FrameNumType. -->
<xsd:simpleType name="FrameNumType">
  <xsd:union memberTypes="b4 b5 ... b15 b16">
    <xsd:annotation>
      <xsd:appinfo>
        <bs2:ifUnion
             value="$SPS/jvt:log2_max_frame_num_minus4 = 0"/>
        <bs2:ifUnion
             value="$SPS/jvt:log2_max_frame_num_minus4 = 1"/>
        <!-- ... -->
        <bs2:ifUnion
             value="$SPS/jvt:log2_max_frame_num_minus4 = 11"/>
        <bs2:ifUnion
             value="$SPS/jvt:log2_max_frame_num_minus4 = 12"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:union>
</xsd:simpleType>
<!-- Use of the FrameNumType data type. -->
<xsd:element name="frame_num" type="jvt:FrameNumType"/>
```

Listing 4.7 illustrates the functioning of the `bs2:ifUnion` construct, used for describing the value space of the `frame_num` syntax element. This parameter is conveyed by the `slice_header()` syntax structure of H.264/AVC. When the first expression evaluates to `true`, type `b4` is selected.

BSDL-0 A number of non-normative extensions are also available in the *Schema for BSDL-0 Extensions*. BSDL-0 contains, among other features, the `bs0:implementation` and `bs0:fillByte` language constructs [110]. The first attribute allows the BS Schema author to provide an alternative implementation of the parsing operations for particular data types, taking the form of user-provided Java classes. This language construct can for instance be used to extend the set of built-in data types that are normatively specified in BSDL-1.

The second non-normative attribute provides a simple byte-alignment mechanism. It instructs a BintoBSD Parser to move to the next byte-aligned position and to place the value of the skipped bits in the corresponding BSD.

The non-normative `bs0:implementation` and `bs0:fillByte` language features were both necessary to describe the Annex B syntax of the first version of the H.264/AVC specification in BSDL (see Section 4.4). In the remainder of this dissertation, BSDL refers not only to the normative tools that are available in BSDL-1 and BSDL-2, but also to the non-normative features specified in BSDL-0. Note that the use of the non-normative BSDL-0 constructs in a BS Schema may affect interoperability among different implementations of the BintoBSD and BSDtoBin processes.

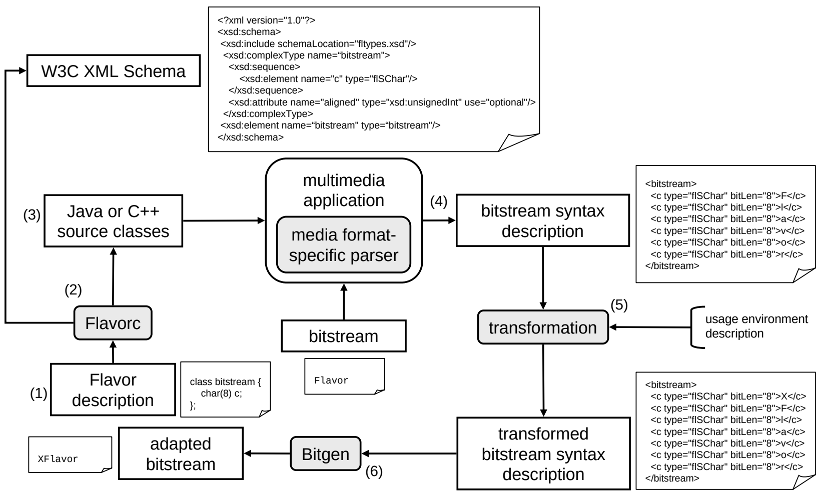
4.3.2 XFlavor

Flavor provides a formal way to specify how data are laid out in a serialized bitstream [111]. It was initially designed as a declarative language with a C++-like syntax to describe the bitstream syntax on a bit-per-bit basis. Its aim is to simplify and speed up the development of software that processes audio-visual bitstreams by automatically generating the required C++ or Java code to parse the data, hence allowing the developer to focus on the processing part of the software. This is possible thanks to the fact that the design of Flavor is based on the principle of separation between bitstream parsing operations and other encoding/decoding operations. This separation does not only acknowledge that the same syntax can be employed by different tool implementations, but also, and even more importantly, that the same tool can work unchanged with a different bitstream syntax. For instance, the number of bits used for a specific quantity can be changed without modifying any part of the decoding algorithms. Finally, unlike the BSD-based approach, a Flavor-based parser does not generate a persistent description of the parsed data, but only an in-memory representation in the form of a collection of C++ or Java class objects.

XFlavor is an extension of Flavor, containing tools for generating an XML description of the entire bitstream syntax and for regenerating an (adapted) bitstream [103]. Figure 4.3 summarizes the overall method for XML-driven media content manipulation by relying on XFlavor, illustrating the editing of a simple character-based bitstream (an extra character ‘X’ is inserted). Explanatory notes are given below:

1. the syntax of a particular media format is described using XFlavor[4];

[4]In the remainder of this dissertation, XFlavor is used as an interchangeable term for Flavor.


W3C XML Schema
(3)
Java or C++ source classes
(2)
Flavorc
(1)
Flavor description
multimedia application
media format-specific parser
(4)
bitstream
Flavor
bitstream syntax description
transformation
(5)
usage environment description
transformed bitstream syntax description
Bitgen
(6)
adapted bitstream
XFlavor

```
<?xml version="1.0"?>
<xsd:schema>
 <xsd:include schemaLocation="fltypes.xsd"/>
  <xsd:complexType name="bitstream">
   <xsd:sequence>
     <xsd:element name="c" type="flSChar"/>
   </xsd:sequence>
   <xsd:attribute name="aligned" type="xsd:unsignedInt" use="optional"/>
  </xsd:complexType>
 <xsd:element name="bitstream" type="bitstream"/>
</xsd:schema>
```

```
class bitstream {
  char(8) c;
};
```

```
<bitstream>
  <c type="flSChar" bitLen="8">F</c>
  <c type="flSChar" bitLen="8">l</c>
  <c type="flSChar" bitLen="8">a</c>
  <c type="flSChar" bitLen="8">v</c>
  <c type="flSChar" bitLen="8">o</c>
  <c type="flSChar" bitLen="8">r</c>
</bitstream>
```

```
<bitstream>
  <c type="flSChar" bitLen="8">X</c>
  <c type="flSChar" bitLen="8">F</c>
  <c type="flSChar" bitLen="8">l</c>
  <c type="flSChar" bitLen="8">a</c>
  <c type="flSChar" bitLen="8">v</c>
  <c type="flSChar" bitLen="8">o</c>
  <c type="flSChar" bitLen="8">r</c>
</bitstream>
```

Figure 4.3: BSD-driven content (i.e., text) adaptation with XFlavor. Note that all bitstream data are embedded in the BSDs.

2. the XFlavor description is translated by Flavorc into an XML Schema (for validation purposes only) and a set of Java or C++ source classes;
3. the source classes, together with a user-provided `main()` method (see Chapter 5), are compiled to a media format-specific parser;
4. BSD creation by the format-specific parser, taking as input a particular bitstream (the BSD is self-containing, i.e. all bitstream data are embedded in the XML description in order to allow different transformations);
5. a BSD is transformed to meet the constraints of a certain usage environment (not meaningful in the context of this artificial example);
6. Bitgen produces an adapted bitstream, only guided by a transformed BSD (as the BSD contains all necessary bitstream data).

Listing 4.8: Flavor fragment for a sub-sequence information SEI message.

```
class Sub_seq_info {
    UnsignedExpGolomb sub_seq_layer_num;
    UnsignedExpGolomb sub_seq_id;
    // first_ref_pic_flag is represented
    // with one bit in the bitstream.
    bit(1) first_ref_pic_flag;
    bit(1) leading_non_ref_pic_flag;
    bit(1) last_pic_flag;
    bit(1) sub_seq_frame_num_flag;
    if ( sub_seq_frame_num_flag )
        UnsignedExpGolomb sub_seq_frame_num;
}
```

Listing 4.9: BSD fragment for a sub-sequence information SEI message.

```
<sub_seq_info>
  <sub_seq_layer_num>
    <!-- ue_code refers to the coded bitstream value. -->
    <ue_code bitLen="1">1</ue_code>
    <!-- ue_value refers to the decoded value. -->
    <ue_value bitLen="0">0</ue_value>
  </sub_seq_layer_num>
  <sub_seq_id>
    <ue_code bitLen="1">1</ue_code>
    <ue_value bitLen="0">0</ue_value>
  </sub_seq_id>
  <first_ref_pic_flag bitLen="1">
    0
  </first_ref_pic_flag>
  <leading_non_ref_pic_flag bitLen="1">
    0
  </leading_non_ref_pic_flag>
  <last_pic_flag bitLen="1">
    0
  </last_pic_flag>
  <sub_seq_frame_num_flag bitLen="1">
    0
  </sub_seq_frame_num_flag>
</sub_seq_info>
```

An XFlavor fragment is provided in Listing 4.8; it describes the syntax of a sub-sequence information SEI message. Listing 4.9 contains a corresponding BSD excerpt, obtained after having parsed a sub-sequence information SEI message using an XFlavor-driven parser. A value of zero for `bitLen` implies that this element is ignored by Bitgen (which is functionality similar to `bs1:ignore` in BSDL).

BSDL defines a number of restrictions and extensions on top of XML Schema. The same observation regarding the use of restrictions and extensions applies to XFlavor, which was designed to be an intuitive and natural extension and restriction of the typing system of object-oriented languages like C++ and

Java [111]. For instance, in XFlavor, in contrast to C++ and Java, it is not possible for an author to define functions, which is due to the declarative nature of XFlavor. However, these design principles of XFlavor will not be discussed in further detail as this dissertation is mainly about the use of XML-based technologies for media resource adaptation. For more information regarding this topic, we would like to refer the interested reader to [111]. Finally, a number of important language features of XFlavor will also be outlined in more detail in Chapter 5.

4.3.3 Summary

Although BSDL and XFlavor share the goal to be applicable to any media format, there are, as discussed by Panis *et al.* [93], some fundamental differences. These differences primarily stem from the differing focus of the technologies.

- In XFlavor, bitstream parsing and description is accomplished by the C++ or Java code generated from the Flavor description. The XML Schema is used only for validation purposes. Proprietary attributes are used in the BSD to indicate the binary encoding of the element content. A drawback of this design is that two elements with the same type (e.g., encoded on two bits) will require the same verbose declaration in the BSD. This is in contrast with BSDL, where the binary encoding of each element is defined once in the BS Schema such that it is no longer necessary to include this encoding information in the resulting BSD.
- In XFlavor, the *complete* bitstream data are actually embedded in the BSD, resulting in potentially verbose descriptions, while BSDL uses a specific data type to point to a data range in the original bitstream when it is too verbose to be included in the BSD (using `bs1:byteRange`).

This is why, unlike XFlavor, BSDL is a description language, rather than a representation language, and can describe the bitstream at a high syntactical level instead of at a low-level, bit-per-bit basis. Finally, note that MPEG-21 BSDL is built on top of W3C XML Schema, which is the metadata community's point of view, while XFlavor is built on top of the principles of object-oriented languages, which is the developers community's point of view.

4.4 BS Schemata for MPEG media formats

In the previous section, we have learned how MPEG-21 BSDL can be used to describe the high-level syntax of a particular media format. The resulting document is called a BS Schema. In this section, a brief overview is given of the most important design characteristics of a number of MPEG-21 BS Schemata that were developed in the context of this research.

XFlavor descriptions were developed as well. However, the construction of these descriptions is not discussed for the following two reasons.

1. The construction of the XFlavor descriptions can be considered a rather rectilinear translation of the tabular representation of the bitstream syntax. Indeed, these tabular representations, typically used in a standards document, are on par with the design principles of XFlavor.

2. XFlavor is a mature language that targets the description of the *entire* bitstream syntax. Consequently, this makes it straightforward to describe *high-level* syntax structures using XFlavor (while the low-level bitstream segments are copied verbatim into the BSD without parsing, in order not to lose any bitstream data; see Chapter 5).

The mature nature of XFlavor is in contrast to the BSDL schema language and its underlying tools. These were characterized by a lack of testing in the context of multiple video coding formats at the time of standardization. This statement will be supported further in this chapter, as well as in Chapter 5.

4.4.1 Structure of the BS Schemata

In Listing 4.10, a generic template is provided for an MPEG-21 BS Schema that describes the structure of a typical packet-based media format. The high-level structure of all media formats, as discussed in this dissertation, is in line with this template. A Parse Unit (PU) is the fundamental unit of processing for a BintoBSD Parser: it may either convey coded header data or coded payload data. For instance, a PU in H.264/AVC equals a NAL unit, while a PU in MPEG-1 Systems can be mapped to a pack.

Listing 4.10: Generic BS Schema for a packet-based media format.

```
<xsd:schema>
  <xsd:element name="bitstream">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parse_unit">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="coded_header_data"/>
              <xsd:element ref="coded_media_data"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Appendix B contains a number of BS Schemata that describe the high-level syntax structures of bitstreams compliant with MPEG-{1, 2} Video, MPEG-{1, 2} Systems, MPEG-4 Visual, and H.264/AVC. The BS Schema for H.264/AVC is discussed in more detail in Section 4.4.3 as its construction required the use of several extensions to BSDL. A BS Schema for VC-1 is outlined in Section 4.5, which gives a complete picture regarding BSD-driven content adaptation using MPEG-21 BSDL and XFlavor. Finally, the construction and use of a BS Schema for H.263+ (including its scalability features) and a preliminary version of SVC is discussed in [34] and [35], respectively.

The main incentive behind the development of our BS Schemata was to test the expressive power of MPEG-21 BSDL, hence their high level of detail[5]. The schemata for the coding formats for instance facilitate the BSD-driven exploitation of temporal scalability, while the BS Schemata for the container formats allow to demultiplex the different elementary bitstreams that are stored in a particular container.

Finally, for the purpose of readability and maintenance, the BS Schemata closely reflect the design principles and the tabular syntax of the media formats described. This implies that the names used in most of the XML element declarations in the BS Schemata are in line with the names employed in the corresponding format specifications. Also, this means that optimized BS Schemata will probably omit some levels in the syntax hierarchy that is used in our BS Schemata.

4.4.2 Complexity assessment of the BS Schemata

In order to get an estimate of the complexity of the BS Schemata, a number of measurements are provided in Table 4.3. These measurements relate to the number of occurrences of the MPEG-21 BSDL language features used in the different BS Schemata, and in particular to the frequency of the attributes that may affect the performance of a BintoBSD Parser. The performance behavior of this parser is discussed at the end of this chapter, as well as in Chapter 5. The semantics and the use of the context management attributes is explained in Appendix C. The measurements pertaining to the different coding and storage

[5]The MPEG-21 DIA specification does not mandate normative BS Schemata.

Table 4.3: Estimate of the complexity of the MPEG-21 BS Schemata, in terms of the number of occurrences of certain language features. This complexity describes the amount of detail in which a particular bitstream may be parsed (and not the complexity of a particular coding format).

language	**MPEG-1**		**MPEG-2**		**MPEG-4**	
feature	**Video**	**Systems**	**Video**	**Systems**	**Visual**	**AVC**
W3C XML Schema components						
`xsd:element`	86	79	137	275	91	323
`xsd:choice`	1	1	1	1	1	1
`xsd:union`	0	0	0	0	1	3
BSDL-2 language features						
`bs2:nOccurs`	0	0	1	1	0	188
`bs2:if`	4	0	5	18	22	82
`bs2:ifUnion`	0	0	0	0	16	29
`bs2:ifNext`	19	11	26	29	8	6
`bs2:length`	0	6	0	30	0	0
`bs2:startCode`	6	6	5	5	3	2
non-normative attributes for context management during BSD generation						
`bs0:startContext`	3	2	4	4	2	32
`bs0:stopContext`	2	1	3	2	2	34
`bs0:partContext`	3	18	4	90	16	64
`bs0:redefineMarker`	1	1	1	2	3	16
other non-normative language features						
`bs0:implementation`	0	0	0	0	0	4
`bs0:fillByte`	1	1	1	1	1	1

formats are done for the BS Schemata submitted to the MPEG meeting in Montreux (2006) [47]. The BS Schema used for H.264/AVC does not contain support for the processing of SEI messages.

4.4.3 Design features of the BS Schema for H.264/AVC

This section provides more details regarding a number of important design features of the BS Schema developed for the first version of H.264/AVC [22] [23]. A good understanding of these design characteristics is for instance fundamental for gaining a good insight into the BS Schema for the scalable extensions to H.264/AVC. Indeed, the BS Schema for SVC, developed by Davy De Schrijver, is a (non-trivial) extension of the BS Schema for the first version of H.264/AVC [35]. It includes, among other features, support for describing the syntax of the scalability information SEI message, which was previously discussed in Chapter 3. Further, the BS Schema for H.264/AVC was also used in the context of several experiments that are discussed in the next two chapters, aiming at the exploitation of multi-layered temporal scalability in H.264/AVC. Finally, the BS Schema for H.264/AVC was also used by Peter Lambert for the XML-based exploitation of ROI scalability in single-layer H.264/AVC [37].

Byte stream NAL unit syntax

Our BS Schema describes the Byte Stream NAL unit syntax of the first version of H.264/AVC, up to and including the syntax of the `slice_header()` structure. As discussed in Chapter 2, the Byte Stream NAL unit format, also known as the Annex B syntax, introduces a delimiter (i.e., a start code prefix) to separate the different NAL units in an elementary H.264/AVC bitstream. This means that the parsing of the high-level syntax of an elementary H.264/AVC bitstream is mainly driven by start codes, and not by length fields[6].

Supported profiles

The BS Schema supports all profiles of the first version of the H.264/AVC standard - this is, the Baseline, Main, and Extended profile. It also allows parsing bitstreams that contain progressive and interlaced coded video. However, the BS Schema for H.264/AVC currently does not have provisions for the profiles as defined by the FRExt amendment, i.e. the so-called High Profiles. FRExt for instance introduces a few additional syntax elements in the different parameter sets.

Parameter set management

The BS Schema for H.264/AVC contains several XPath expressions, primarily used to get access to information stored in the different types of parameter sets. These XPath expressions provide support for H.264/AVC bitstreams that contain multiple SPSs and PPSs. This feature is for example necessary in case of H.264/AVC bitstreams that support changing macroblock allocation maps, a bitstream configuration that may occur when FMO is in use. Indeed, a change in the macroblock allocation map is signaled by sending a new PPS.

Also, our XPath expressions guarantee a correct access to the currently active SPS and PPS, even when those parameter sets are characterized by the same identifier. This may for example be the case when a parameter set is repeatedly sent within an H.264/AVC bitstream for error resilience purposes. The identifier for the PPS that is currently active is stored in the `jvt:pic_parameter_set_id` syntax element in a slice header. On the other hand, the identifier for the SPS that is currently active is stored in the `jvt:seq_parameter_set_id` syntax element in the currently activated PPS.

[6]Length fields can for instance be used to skip the payload of irrelevant SEI messages, as the payload length of an SEI message is signaled by a syntax element that precedes the payload syntax structure (see Figure B.13).

The semantics and use of our XPath expressions is explained using an example. The number of bits needed to represent the syntax element `jvt:frame_num` in the `slice_header()` syntax structure is dependent on the value of the `jvt:log2_max_frame_num_minus4` syntax element in the SPS that is currently active. As shown in Listing 4.7, the exact number of bits for the representation of `frame_num` is determined using an `xsd:union` construct. The first XPath expression that is employed in this `xsd:union` construct is completely provided in Listing 4.11. As explained below, this XPath expression can be used in the context of H.264/AVC bitstreams containing multiple SPSs and multiple PPSs.

Listing 4.11: First XPath expression used in the `xsd:union` construct for `frame_num` (split up in three different parts for readability purposes).

```
<bs2:ifUnion value="/jvt:bitstream/jvt:byte_stream/
jvt:byte_stream_nal_unit[$PH1][position()=last()]/jvt:nal_unit/
jvt:raw_byte_sequence_payload/jvt:seq_parameter_set_rbsp/
jvt:log2_max_frame_num_minus4 = 0"/>

<!-- Retrieve the active SPS. -->
$PH1 = jvt:nal_unit/jvt:raw_byte_sequence_payload/
jvt:seq_parameter_set_rbsp/jvt:seq_parameter_set_id = $PH2/
jvt:nal_unit/jvt:raw_byte_sequence_payload/
jvt:pic_parameter_set_rbsp/jvt:seq_parameter_set_id

<!-- Retrieve the active PPS. -->
$PH2 = /jvt:bitstream/jvt:byte_stream/jvt:byte_stream_nal_unit
[jvt:nal_unit/jvt:raw_byte_sequence_payload/
jvt:pic_parameter_set_rbsp/jvt:pic_parameter_set_id =
/jvt:bitstream/jvt:byte_stream/jvt:byte_stream_nal_unit
[position()=last()]/jvt:nal_unit/
jvt:raw_byte_sequence_payload/child::*/child::*/
jvt:slice_header/jvt:pic_parameter_set_id][position()=last()]
```

The XPath expression shown in Listing 4.11 reflects the pointer-based relationship between a slice header, a PPS, and an SPS. This expression is used to navigate from within a particular slice header to the value of the `jvt:log2_max_frame_num_minus4` syntax element, conveyed by a particular SPS. The semantics of the XPath expression is as follows (from a decoder point of view). Based on the PPS identifier that is carried by a slice header, the correct PPS is activated and accessed. This is implemented by the expression labeled `$PH2` (placeholder 2). In a next step, the SPS identifier, carried by the PPS that is currently active, is used to activate and to access the correct SPS. This is realized using the expression labeled `$PH1` (placeholder 1). Finally, the `jvt:log2_max_frame_num_minus4` syntax element is retrieved from the appropriate SPS to verify whether its value is equal to zero (implying that `frame_num` is represented with four bits in the bitstream).

Listing 4.12 shows a simplified XPath expression that can be employed as the first entry in the `xsd:union` construct for `frame_num` in case of an H.264/AVC bitstream that is only using one SPS and one or multiple PPSs. This XPath expression does no longer contain predicates. The length of the location step is reduced significantly.

Listing 4.12: Simplified XPath expression used in the `xsd:union` construct for `frame_num`.

```
<bs2:ifUnion value="/jvt:bitstream/jvt:byte_stream/
jvt:byte_stream_nal_unit_sps/jvt:nal_unit_sps/
jvt:raw_byte_sequence_payload_sps/jvt:seq_parameter_set_rbsp/
jvt:log2_max_frame_num_minus4 = 0"/>
```

MPEG-21 BSDL extensions used

The design of the BS Schema for H.264/AVC, describing its Annex B syntax with a granularity of up to and including the `slice_header()` syntax structure, required the use of a number of extensions to the MPEG-21 BSDL specification. At the time of developing this BS Schema, the BSDL reference software already provided the necessary non-normative means to incorporate our extensions in the BS Schema for H.264/AVC.

Use of `bs0:fillByte`. The development of the BS Schema for H.264/AVC required the use of the `bs0:fillByte` language construct. This attribute instructs the BintoBSD Parser, as available in the reference software package, to search for the next byte-aligned position. The value of the bits skipped is placed in the BSD. After the transformation of the BSD, the value of the skipped bits is subsequently used by the BSDtoBin process to add an appropriate amount of padding bits to the output bitstream. Note that the number of padding bits might have changed due to changes applied to a BSD (in particular when syntax elements, represented with variable-length coding, were removed, added, or changed). Zero-valued bits are added to the output bitstream in case the number of required output padding bits surpasses the number of bits skipped by BintoBSD. The use of `bs0:fillByte` is illustrated by Listing 4.13.

Listing 4.13: Use of `bs0:fillByte`.

```
<!-- BS Schema fragment. -->
<xsd:simpleType name="Stuffing">
  <xsd:restriction base="bs0:fillByte"/>
</xsd:simpleType>
<xsd:element name="bit_stuffing" minOccurs="0" maxOccurs="1"
     type="jvt:Stuffing"/>
<!-- BSD fragment. -->
<bit_stuffing>15</bit_stuffing>
```

The `bs0:fillByte` attribute allows limiting the overhead of the amount of XML data produced when describing the high-level syntax of a bitstream. As such, its use contributes to the scalable nature of BSDs. Without the availability of the `bs0:fillByte` language construct, it is necessary to parse the bitstream to a position that is known for its byte-alignment, despite the fact that the information processed is not relevant for the purpose of content adaptation. This is due to the way parsing is defined for the BintoBSD process: the instruction to search for a next start code (delimiter-driven parsing) or the instruction to skip a well-defined number of bytes (length-driven parsing) can only be sent to a BintoBSD Parser at a byte-aligned position.

Acquiring byte-alignment can be a major challenge in case of high-level syntax structures (e.g., headers) that are synthesized using syntax elements that are either optional or that are either represented using variable-length coding. This is for instance true for most of the headers employed in an H.264/AVC bitstream: several high-level parameters are encoded using Exponential Golomb coding, a variable-length coding scheme. For such syntax structures, byte-alignment can often only be achieved by parsing the entire syntax structure (when no `bs0:fillByte` language construct is available). It is clear that this approach fails in realizing the idea to create a high-level BSD for a particular bitstream.

The implementation of the `bs0:fillByte` language feature, as currently provided in the reference software package, may also result in a number of unexpected side effects when doing editing operations on syntax elements represented by a variable-length coding scheme. More precisely, in case of a shortage of bits during bitstream regeneration, by default, zero-valued bits are added to the newly generated bitstream. However, in some cases, it is mandatory to use padding bits that are having a value of one (or to use other specific bit patterns, such as the frequently used 0(1)* bit pattern in MPEG-4 Visual and the 1(0)* bit pattern in H.264/AVC). For these cases, the use of the `bs0:fillByte` construct may result in the creation of evil bitstreams [24].

The bs0:fillByte attribute maps to byte_aligned(), a syntax function that can be found in the H.264/AVC specification, although the semantics are not completely the same (e.g., byte_aligned() is only used for determining byte-alignment, while bs0:fillByte can also be used to adjust byte-alignment by inserting additional bits in the bitstream). As shown in Table 4.3, the use of the non-normative bs0:fillByte language construct was necessary in the BS Schemata for other media formats as well.

Use of bs0:implementation. Besides the use of bs0:fillByte, the design of the BS Schema for H.264/AVC also required the employment of the bs0:implementation language construct. This non-normative language construct adds a programming language dependent extension mechanism to BSDL. Indeed, the bs0:implementation attribute makes it possible to rely on procedural objects to perform complex computations or to process complex data types. Complex means that it is not trivial in practice to express the computations using BSDL, or to describe the layout of a particular data type using the aforementioned schema language.

In the BSDL reference software, the procedural objects take the form of user-provided Java class files. Listing 4.14 illustrates how to address such a class file from within a BS Schema: the bs0:implementation attribute takes a path, referring to a particular Java class file, as value. As such, the set of BSDL-1 built-in data types is extended with support for a new data type named UnsignedExpGolomb.

Listing 4.14: Use of bs0:implementation.

```
<!-- BS Schema fragment. -->
<xsd:simpleType name="UnsignedExpGolomb" bs0:implementation=
    "datatypes.UnsignedExpGolomb">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<!-- BSD fragment. -->
<xsd:element name="slice_type" type="jvt:UnsignedExpGolomb"/>
```

The Java classes, residing in the Java class files, have to implement a well-defined interface.

- A first method, called readSCFromBitstream(), is used by BintoBSD to retrieve a particular number of bits from the bitstream. After having processed these bits, the resulting value is written to a BSD.

- A second method, called `writeSCToBitstream()`, is used by BSDtoBin to convert a value, stored in the BSD, to a bit pattern that is subsequently written to the bitstream.

The following Java classes were developed in order to successfully create a BS Schema for the first version of H.264/AVC.

- `UnsignedExpGolomb` - This class is used for processing syntax elements that are represented by an Unsigned Exp-Golomb code. For BintoBSD, the class retrieves a bit string from the bitstream and subsequently writes the decoded value to the BSD. For BSDtoBin, the value in the BSD is transformed by the class into an appropriate bit pattern that is written to the bitstream in a next step.

  Note that the construction of Unsigned Exp-Golomb codes can be described using BSDL. However, this approach results in a tremendous overhead: a separate XML element has to be created for every bit that is part of the syntax element represented by an Unsigned Exponential Golomb code. On top of that, it is not trivial to decode or to interpret the resulting representation of such a syntax element in XML using an XPath expression.

- `SignedExpGolomb` - Similar to the `UnsignedExpGolomb` class, the `SignedExpGolomb` class is used for processing syntax elements represented by a Signed Exp-Golomb code.

- `SliceGroupChangeCycleType` - This class is used for the processing of the `jvt:slice_group_change_cycle` syntax element, which is the last syntax element of the `slice_header()` syntax structure when FMO types 3, 4, or 5 are in use (evolving slice groups). It determines the number of macroblocks in slice group 0. The use of the `bs0:implementation` language construct is necessary because the number of bits needed for the representation of the `jvt:slice_group_change_cycle` syntax element has to be computed by evaluating a logarithmic function with base two:

  $$Ceil(Log_2(\frac{PicSizeInMapUnits}{SliceGroupChangeRate} + 1)). \tag{4.1}$$

  In the context of bitstream parsing, the logarithmic function with base two is typically used for determining the number of bits a syntax element is represented with, the exact number of bits dependent on the value of another syntax element that is given as argument to the logarithmic function. However, the logarithmic function is not available in the XPath 1.0 specification.

When the set of input values is limited, the use of a logarithmic function can be circumvented using the `xsd:union` language feature and values that are computed beforehand. However, this is not the case for the `jvt:slice_group_change_cycle` syntax element because the number of input values is dependent on the picture resolution.

Finally, the `bs2:length` facet only allows expressing the length of a bitstream segment in terms of a number of bytes. As such, the processing of `jvt:slice_group_change_cycle` would still pose a problem in case a logarithmic function would have been available in XPath 1.0. Indeed, the first edition of MPEG-21 BSDL does not offer support for data types that are characterized by a variable bit length.

- `CabacAlignmentOneBitType` - This class is used for the processing of the `jvt:cabac_alignment_one_bit` syntax element, which may occur as the first syntax element in the `slice_data()` syntax structure when the CABAC entropy coding scheme is in use.

  When CABAC is employed, the H.264/AVC standard requires that byte-alignment is achieved after having parsed the `slice_header()` syntax structure. This is realized by starting the `slice_data()` syntax structure with an appropriate number of `jvt:cabac_alignment_one_bit` syntax elements. The `jvt:cabac_alignment_one_bit` syntax element has a length of one bit, which always has a value of one.

  The use of the `CabacAlignmentOneBitType` class is necessary when the size of the `slice_header()` structure is changed due to editing operations on syntax elements represented by a variable-length coding scheme. During the construction of a bitstream by BSDtoBin, the `CabacAlignmentOneBitType` class adds a sufficient number of one-valued bits to the output bitstream such that a correct transition is guaranteed between the `slice_header()` and `slice_data()` syntax structures.

  Note that `bs0:fillByte` cannot be used for this purpose, as this attribute does not allow specifying that bits with a value of one have to be used to achieve byte-alignment; the use of zero-valued padding bits would lead to evil bitstreams in this particular case.

- `payloadTypeType` - This class is used to determine the payload type of SEI messages. This is something that can also be expressed in BSDL. However, the employment of this class allows the use of a number of straightforward XPath expressions, as such improving the readability of our BS Schema.

- `payloadSizeType` - Similar to the `payloadTypeType` class, the `payloadSizeType` class is used to determine the payload size of an SEI message, leading to more elegant XPath expressions in the BS Schema.

Finally, a NAL unit may carry multiple SEI messages. To detect the different SEI messages in the payload of such a NAL unit, the H.264/AVC standard mandates the use of a syntax function called `more_rbsp_data()`. This function also needs to be implemented using the `bs0:implementation` extension mechanism. However, as this research only focuses on the use of NAL units containing one SEI message, the implementation of `more_rbsp_data()` will not be discussed in more detail.

4.5 BSD-driven temporal adaptation in VC-1

In the previous sections, we have outlined two languages that enable BSD-driven adaptation of binary media resources - this is, MPEG-21 BSDL and XFlavor. In this section, a discussion is provided regarding a performance evaluation of two practical frameworks for BSD-driven content adaptation, based on the aforementioned languages: one framework is entirely built on top of MPEG-21 BSDL while the second framework is completely relying on the use of XFlavor. Both infrastructures are employed for the adaptation of VC-1 compliant bitstreams in the temporal domain. The use of VC-1 in the next series of experiments is justified for the following three reasons:

1. VC-1 is a state-of-the-art video coding format that can be seen as the main competitor of H.264/AVC in terms of coding efficiency, next to the Chinese Audio and Video coding Standard (AVS) specification [112];

2. it is demonstrated that MPEG-21 BSDL can be applied in a straightforward way to VC-1 for the purpose of BSD-driven exploitation of temporal scalability;

3. an insight into the coding structure of VC-1's Advanced Profile, which is similar to the coding structure of H.264/AVC (see below), will ease some of the discussions in the next two chapters.

A complete overview is provided, involving BSD generation, BSD transformation, and tailored bitstream generation. The performance measurements, which are also published in [28], are given in terms of computational times, file sizes, and memory consumption. Our analysis is not intended to be exhaustive: it is rather to be considered a first acquaintance for the interested reader with a number of performance bottlenecks in the two different approaches toward BSD-driven content adaptation, using an emerging video coding format. A more thorough discussion regarding the identified bottlenecks will be provided in the subsequent chapters for the MPEG-1 Video and H.264/AVC coding formats, together with a number of appropriate solutions or alternatives.

4.5.1 Video Codec 1

In August 2005, VC-1 reached the Final Committee Draft (FCD) status with SMPTE's C24 Technology Committee, in which it is officially referenced as SMPTE standard 421M [94]. This specification for digital video coding not only sits at the core of the Windows Media Series [113] (Windows Media Video 9 is Microsoft's implementation of VC-1[7]), but it is also included as a mandatory video compression format for the next-generation of High-Definition DVDs by both the Blu-Ray Disc Association and the DVD Forum (together with H.262/MPEG-2 Video and H.264/AVC). Hence, it is likely that VC-1 will be used in diverse usage environments, thus making it relevant to gain an insight into techniques that allow to realize an efficient and transparent customization of VC-1 compliant bitstreams.

SMPTE's VC-1 is an emerging standard for the coding of digital consumer video: pictures are represented in the YCbCr color space with 4:2:0 sampling, using eight bits per color component. The VC-1 specification defines three profiles.

- The **Simple Profile** targets low-rate Internet streaming and low-complexity applications such as video playback on Personal Digital Assistants (PDAs). This profile can be considered equivalent to the Simple Profile of MPEG-4 Visual.
- The **Main Profile** aims at high-rate Internet applications such as TV/Video-On-Demand over IP. This profile can be considered equivalent to the Advanced Simple Profile of MPEG-4 Visual.
- The **Advanced Profile (AP)** focuses on broadcast applications, such as Digital TV, HD DVD for PC playback, or HDTV. It is the only profile

[7] `http://www.microsoft.com/windows/windowsmedia/forpros/events/NAB2005/VC-1.aspx`.

that supports interlaced content and the use of slices. This profile can be considered equivalent to the High Profile of H.264/AVC.

Bitstreams compliant with the AP are self-contained; their decoding is not dependent on information that has to be conveyed by an external transport mechanism such as a file container. The latter observation is not true for bitstreams that are in line with the Simple and Main profile. Their decoding requires Decoder Initialization Metadata (DIM). These metadata items have to be made available to a decoder prior to the start of the decoding process. For instance, the profile and level used have to be communicated to a decoder by external means in case of the Simple and Main profile, while this information is readily available for a decoder in case of AP-compliant bitstreams. Hence, this explains why we have chosen to only describe the high-level structure of bitstreams that are satisfying the constraints of VC-1's most sophisticated profile.

A VC-1 bitstream consists of a number of Encapsulated Bitstream Data Units (EBDUs). These units, which are similar to NAL units in H.264/AVC or to frame data units in Dirac[8], may carry compressed picture data (frame and slice EBDUs), as well as header information (sequence and entry-point header EBDUs). A simplified overview of the high-level structure of a VC-1 bitstream, in line with the possibilities of the AP, is provided in Figure 4.4.

In a VC-1 AP bitstream, the first slice of a frame is conveyed by a frame EBDU, while the remaining slices of the frame are stored in a separate slice EBDU. This is similar to the way slices are stored in NAL units in H.264/AVC. However, in the context of BSD-driven temporal adaptation of VC-1 bitstreams, using BSDL and XFlavor, a frame EBDU acts as a container for all the slice EBDUs of a particular frame - this is, in a BSD for an AP bitstream, a `smpte:slice` element will be child of a `smpte:frame` element. This is for instance illustrated by Listing 4.15, showing a BSD produced by BSDL's BintoBSD Parser. Our motivation is twofold:

1. the decision-making of the BSD transformation engine regarding which frame to drop can be done at the level of a frame, and not at the level of the slices (i.e., not at the level of the individual EBDUs); and

2. the XSLT stylesheets, which are responsible for dropping certain types of frames at the level of a BSD, can be kept simple and uniform.

[8]As such, a `parse_unit` in our generic BS Schema template (see Listing 4.10) corresponds to an EBDU for VC-1.

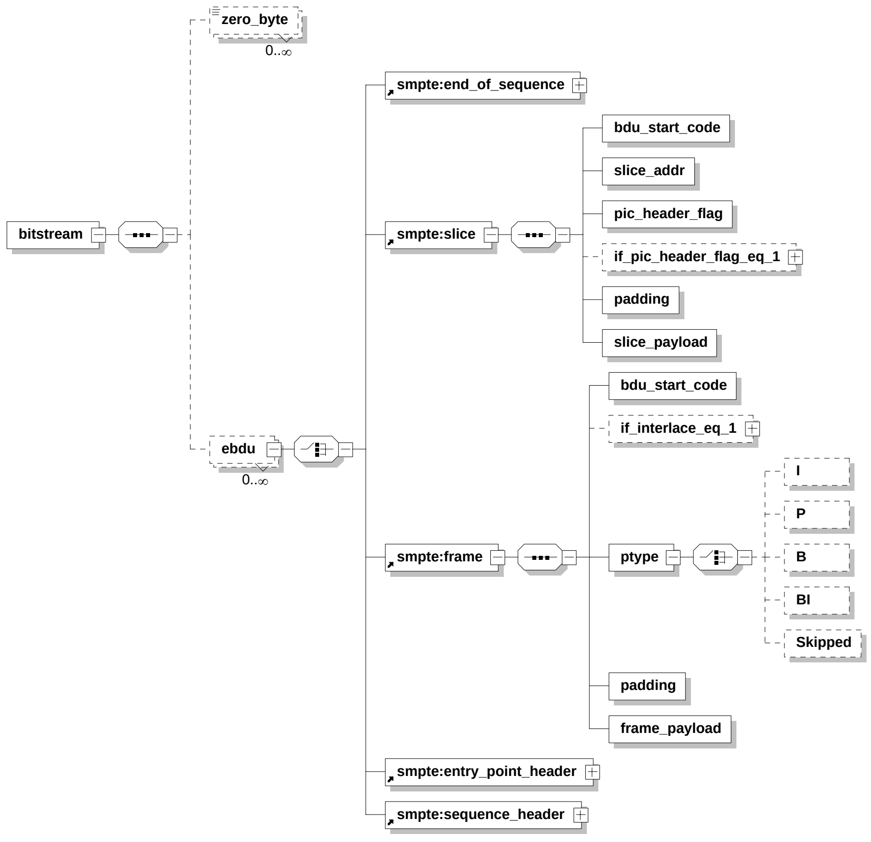


Figure 4.4: Simplified overview of the VC-1 bitstream structure. For this BS Schema, a slice EBDU sits at the same level as a frame EBDU.

Listing 4.15: Simplified BSD produced by BintoBSD. All slices of one particular picture share the same coding type.

```
<bitstream bs1:bitstreamURI="./SA10098.vc1">
  <encapsulated_bdu>
    <sequence_header>
      <bdu_start_code>0000010F</bdu_start_code>
      <profile>3</profile>
      <level>2</level>
      <colordiff_format>1</colordiff_format>
      <frmrtq_postproc>7</frmrtq_postproc>
      <btrtq_postproc>31</btrtq_postproc>
      <postprocflag>0</postprocflag>
      <max_coded_width>359</max_coded_width>
      <max_coded_height>239</max_coded_height>
      <!-- ... -->
    </sequence_header>
  </encapsulated_bdu>
  <encapsulated_bdu>
    <entry_point_header>
      <bdu_start_code>0000010E</bdu_start_code>
      <!-- ... -->
    </entry_point_header>
  </encapsulated_bdu>
  <encapsulated_bdu>
    <!-- A frame EBDU. -->
    <frame>
      <bdu_start_code>0000010D</bdu_start_code>
      <ptype>
        <I>6</I>
      </ptype>
      <padding>0</padding>
      <frame_payload>143 17753</frame_payload>
      <!-- A slice EBDU. -->
      <child_ebdu>
        <slice>
          <bdu_start_code>0000010B</bdu_start_code>
          <slice_addr>8</slice_addr>
          <pic_header_flag>1</pic_header_flag>
          <if_pic_header_flag_eq_1>
            <ptype>
              <I>6</I>
            </ptype>
          </if_pic_header_flag_eq_1>
          <padding>0</padding>
          <slice_payload>17902 62560</slice_payload>
        </slice>
      </child_ebdu>
      <!-- ... -->
    </frame
  </encapsulated_bdu>
  <!-- ... -->
</bitstream>
```

A sequence-level header contains parameters that are used to decode a sequence of compressed pictures. The entry-point header serves two purposes. First, it is used to signal a random access point: it guarantees that subsequent pictures can be decoded (closed GOP functionality). Second, it is used to signal changes in the coding control parameters that are enabled for a particular entry point segment. In contrast to the parameter sets in H.264/AVC, sequence-level headers and entry-point headers are immediately activated when retrieving them from an elementary bitstream.

Further, the VC-1 specification also defines five types of pictures.

- An **I picture** (intra-coded picture) is coded using information only from itself; all its macroblocks are intra-coded.
- A **P picture** is coded using MCP from past reference pictures. It may contain macroblocks that are inter- or intra-coded.
- A **B picture** is coded using MCP from past and/or future reference pictures; macroblocks can be inter- or intra-coded.
- A **BI picture** is a B picture that only contains intra-coded macroblocks. It cannot be used for predicting other pictures (see further). In other words, a BI picture is a non-reference I picture.
- A **Skipped picture** is a P picture that is identical to its reference picture; its reconstruction is equivalent to copying the reference picture, implying that no further data are transmitted. This is similar to the use of pseudo-skipped pictures in H.262/MPEG-2 Video [114] and Non-coded Video Object Planes (N-VOPs) in MPEG-4 Visual (see also Chapter 6).

B pictures in VC-1 are not used as a reference for subsequent pictures. They are placed outside the decoding loop, allowing shortcuts to be taken during their decoding without causing drift or long-term visual artifacts (temporal scalability). Intra-coded B pictures (BI pictures) are also allowed in VC-1. They typically occur at scene changes where it is more economical to code the data as intra rather than P or B. This picture type is distinguished from true I pictures by disallowing them to be referenced by other pictures. This allows a decoder (e.g., on a constrained device) to omit decoding them. It also allows an encoder to choose a lower quality setting or quantization step size to encode them. Finally, an AP bitstream offers a trivial form of spatial scalability: the coded size in pixels can be changed at entry points. This provides an encoder with the ability to alter the coded picture size and thus the bit rate.

Listing 4.16: XSLT stylesheet for frame dropping in a BSD, produced by BintoBSD for a particular VC-1 bitstream.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:smpte="vc-1">
  <xsl:output method="xml" indent="yes"/>
  <!-- Match all. -->
  <xsl:template name="tplAll" match="@*|node()">
    <xsl:copy>
```

```
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <!-- Match top-level element. -->
  <xsl:template match="smpte:bitstream">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <!-- Do nothing for processing instructions. -->
  <xsl:template match="processing-instruction()"/>
  <!-- Drop B pictures - Overrides tplAll. -->
  <xsl:template name="tplB_picture"
       match="smpte:encapsulated_bdu
             [smpte:frame/smpte:ptype/smpte:B=2]">
    <!-- Do nothing. -->
  </xsl:template>
  <!-- Drop BI pictures - Overrides tplAll. -->
  <xsl:template name="tplBI_picture"
       match="smpte:encapsulated_bdu
             [smpte:frame/smpte:ptype/smpte:BI=14]">
    <!-- Do nothing. -->
  </xsl:template>
  <!-- Drop Skipped pictures - Overrides tplAll. -->
  <xsl:template name="tplSkipped_picture"
       match="smpte:encapsulated_bdu
             [smpte:frame/smpte:ptype/smpte:Skipped=15]">
    <!-- Do nothing. -->
  </xsl:template>
</xsl:stylesheet>
```

4.5.2 Performance data

In this section, a number of performance measurements are presented for the BSDL and XFlavor content adaptation chains. An off line scenario is targeted that requires the elimination of B pictures, BI pictures, and Skipped pictures in bitstreams compliant with VC-1's AP. Other use cases in the compressed temporal domain can be devised as well, such as content summarization and scene selection.

An artificial set of test bitstreams was generated from a representative conformance test bitstream (SA10098.vc1; AP@L1; 6000 Kbps; 720x480; 30 Hz; slice coding enabled), using the BSD approach[9]. This is due to the fact that a

[9]BSDs can be easily used for straightforward editing operations such as the removal or duplication of a number of frames.

Table 4.4: VC-1 bitstream characteristics.

		#headers		#frames[a]			
#frames	#EBDUs	sequence	entry-point	I	P	B	Skipped
150	1866	1	2	1	41	74	34
301	3531	1	2	1	79	150	71
450	5244	1	2	2	119	224	105
601	7084	1	2	2	155	300	144
750	8762	1	3	3	186	374	187
900	10498	1	3	3	225	449	223
1800	20994	1	6	6	450	898	446
2700	31490	1	8	9	675	1347	669
3600	41986	1	12	12	900	1796	892
4500	52482	1	15	15	1125	2245	1115
9000	104962	1	30	30	2250	4490	2230

[a]The test bitstreams do not contain BI pictures.

software encoder, able to output elementary bitstreams, was not at our disposal at the time of doing this research; only a reference decoder and a set of conformance bitstreams were available in the public domain. The characteristics of the test bitstreams used are summarized in Table 4.4.

The measurements were done on a PC having an Intel Pentium M 1.6 GHz CPU and 512 MB of system memory at its disposal. SAXON8[10] was used in order to apply XSLT stylesheets to BSDs. The BSDs are either produced by BSDL's BintoBSD Parser or by a parser that is automatically generated using XFlavor. Version 1.2.1 of the MPEG-21 BSDL reference software was used[11], as well as version 5.1.0 of the Flavorc translator[12]. The peak heap memory consumption was registered by relying on JProfiler 4.0.2[13].

The XSLT stylesheets make it possible to eliminate the descriptions of B, BI, and Skipped pictures in the XML domain. Such a stylesheet is for instance provided in Listing 4.16; it allows the transformation of BSDs that are produced by BSDL's BintoBSD Parser. A similar stylesheet was used for the transformation of BSDs created by the XFlavor-based parser. The most important observations will now be put forward.

[10]Available online: `http://saxon.sourceforge.net/`.

[11]Available online: `http://www.enikos.com/mpegarea/`.

[12]Available online: `http://flavor.sourceforge.net/`.

[13]Available online: `http://www.ej-technologies.com/products/jprofiler/overview.html`.

Table 4.5: Speeds for BSD generation, BSD transformation (using XSLT), and bitstream generation.

	BSD generation[a]		BSD transformation		bitstream generation	
#frames	BintoBSD (EBDUs/s)	XFlavor (EBDUs/s)	BSDL (frames/s)	XFlavor (frames/s)	BSDtoBin (EBDUs/s)	Bitgen (EBDUs/s)
150	24.4	162.5	316.3	2.5	962.2	42.0
301	15.7	163.7	455.4	0.4	1569.9	41.3
450	11.6	181.8	488.3	n/a	2140.4	n/a
601	9.0	189.5	564.4	n/a	2318.5	n/a
750	7.5	174.2	649.4	n/a	2679.2	n/a
900	6.4	169.0	610.0	n/a	2828.0	n/a
1800	3.3	153.3	771.5	n/a	3531.4	n/a
2700	1.3	164.5	940.0	n/a	3084.2	n/a
3600	n/a	186.2	n/a	n/a	n/a	n/a
4500	n/a	186.1	n/a	n/a	n/a	n/a
9000	n/a	172.6	n/a	n/a	n/a	n/a

[a]Certain data items are missing due to performance issues, indicated by n/a (not available).

Table 4.5 contains the speeds obtained for the creation of BSDs, their transformation, and for the subsequent construction of adapted VC-1 bitstreams. Regarding the different BSD generation techniques, it is clear that BSDL's BintoBSD Parser cannot be used in practice. This parser is characterized by a decreasing processing speed and an increasing memory consumption in function of the bitstream length (see Table 4.5 and Table 4.7). As extensively discussed in the next chapter, this is due to the fact that the entire BSD needs to be kept in the system memory to allow the evaluation of arbitrary XPath 1.0 expressions. The speeds obtained by the XFlavor-based parser are satisfactory: a throughput of at least 153 EBDUs per second can be achieved.

Table 4.5 also summarizes the speeds obtained for the transformation of the different BSDs and for the derivation of tailored VC-1 bitstreams. The measurements show that the transformation of XFlavor BSDs fails in this regard. As illustrated by Table 4.6, this is due to the verboseness of the textual BSDs produced by the XFlavor-based parser: all bitstream data are embedded in the BSDs in order not to lose any bitstream data and to allow different transformations of the BSDs (see the next chapter for more technical details). On the other hand, the high-level BSDs that are produced by the BintoBSD Parser can be transformed at a speed that is faster than real time. BSDL's format-agnostic BSDtoBin Parser, built on top of the Simple API for XML (SAX) and using buffered I/O, also operates at a speed that is faster than real time.

Table 4.6: Bitstream and textual BSD sizes, expressed in megabytes (MB).

	original file sizes			transformed file sizes		
#frames	bitstream	BSDL	XFlavor	BSDL	XFlavor	bitstream
150	4.2	0.5	88.7	0.3	27.9	1.3
301	7.7	1.0	161.9	0.5	47.3	2.3
450	10.9	1.5	228.1	0.7	n/a	3.2
601	14.1	2.0	295.1	0.9	n/a	4.0
750	17.5	2.5	366.8	1.1	n/a	4.9
900	21.1	3.0	442.1	1.4	n/a	6.0
1800	42.1	6.0	884.2	2.7	n/a	12.0
2700	63.2	9.1	1326.3	4.1	n/a	18.0
3600	84.2	n/a	1768.5	n/a	n/a	n/a
4500	105.3	n/a	2210.6	n/a	n/a	n/a
9000	210.6	n/a	4421.1	n/a	n/a	n/a

Further, Table 4.7 provides an overview of the memory consumption of the different tools involved. As previously mentioned, BSDL's BintoBSD Parser is characterized by an increasing memory usage, while the XFlavor-based parser for VC-1 demonstrates an almost constant and low peak memory usage. The XSLT software also requires an increasing amount of memory: the XSLT transformation engine needs to build up an entire Document Object Model (DOM) tree in order to be able to execute the required transformations. This illustrates the necessity to pay attention to the transformation technology used in practical situations, an observation that is confirmed by the research results presented in [16]. Finally, BSDL's BSDtoBin Parser shows a low and constant memory usage.

4.6 Conclusions and original contributions

In this chapter, we first introduced the principles of BSD-driven media resource adaptation. Next, two languages were reviewed for describing the (high-level) syntax of binary media resources, in particular MPEG-21 BSDL and Flavor, extended with XML features (XFlavor). Although created from a different point of view, both languages offer means for exposing the structure of a binary media resource as an XML-based BSD, and for generating a tailored media resource using a transformed BSD.

The discussion of the different bitstream syntax description languages emphasized the use of BSDL, for reasons mentioned in the text. MPEG-21 BSDL aims at the creation of a generic software architecture for format-agnostic con-

Table 4.7: Peak heap memory consumption, expressed in megabyte (MB).

	BSD generation		BSD transformation	bitstream generation
#frames	BintoBSD	XFlavor	XSLT	BSDtoBin
150	6.6	0.7	3.2	1.0
301	12.2	0.8	4.8	0.9
450	19.0	0.8	8.9	1.2
601	33.8	0.8	8.7	1.2
750	40.0	0.9	8.6	1.3
900	61.0	0.9	16.9	1.3
1800	65.0	0.9	34.6	1.3
2700	n/a	1.4	43.3	1.4
3600	n/a	1.6	n/a	n/a
4500	n/a	1.8	n/a	n/a
9000	n/a	1.9	n/a	n/a

tent adaptation, and of which the operation is entirely steered by XML-based technologies. This concept is similar to how XHTML and Cascading Style Sheet (CSS) documents guide a browser to render a web page. While scalable media formats (content) are providing a solution for dealing with the heterogeneity in present-day terminals and networks (context), BSDL is offering a format-agnostic solution for the actual adaptation of scalable media resources. Consequently, BSDL is a solution for dealing with an increasing number of (scalable) media formats, used in heterogeneous usage environments.

A first original contribution, described in this chapter, consists of testing the expressive power of BSDL by defining fine-grained BS Schemata for a number of video coding and container formats: MPEG-1 Video, MPEG-1 Systems, H.262/MPEG-2 Video, MPEG-2 Systems (Program Streams and Transport Streams), MPEG-4 Visual, H.264/AVC, and VC-1. The design of these BS Schemata is in line with a generic template for packet-based media formats, which was identified in the course of this research. The development of a BS Schema for the first version of H.264/AVC, which is at the foundation of a BS Schema for SVC, can be considered a non-trivial achievement as it required the use of several non-normative extensions to the MPEG-21 BSDL schema language. These enhancements were for instance needed to achieve byte-alignment functionality and to introduce a number of new data types in BSDL. Note that our BS Schema for H.264/AVC facilitates the exploitation of different types of temporal scalability in H.264/AVC, as well as of region of interest extraction (see Chapter 6).

The high-level syntax of most of the aforementioned media formats was also described using XFlavor. The construction of these syntax descriptions can be considered rather straightforward. This is mainly due to the expressive power of the language in question, which aims at the description of the entire bitstream syntax.

Besides testing the expressive power of MPEG-21 BSDL and XFlavor, we also evaluated their performance in the context of the media formats described, targeting applications such as BSD-driven temporal adaptation and demultiplexing (see for instance the stylesheets in Listing D.6 and Listing D.7 in Appendix D). This performance assessment of BSDL and XFlavor was realized by breaking up the functioning of the different tool chains for media resource adaptation in their three fundamental building blocks: BSD generation, BSD transformation, and tailored bitstream generation using a transformed BSD. In this chapter, we discussed a representative performance analysis regarding the BSD-driven exploitation of temporal scalability in VC-1 compliant bitstreams. This resulted in the identification of a number of bottlenecks, in particular the slow and memory-consuming generation of BSDs by BSDL's BintoBSD Parser, the verbose BSDs produced by XFlavor-based parsers, and the memory-consuming transformation of BSDs using XSLT.

In Chapter 5, we will demonstrate that the inefficient BSD generation in a BSDL-based content adaptation chain is a fundamental problem of the BintoBSD process, implying that these performance issues cannot merely be solved by software optimizations. We will also propose an alternative for the efficient generation of BSDs compliant with MPEG-21 BSDL. This solution is based on a modification of XFlavor. Finally, in Chapter 6, we will briefly outline a solution for the efficient transformation of large textual BSDs. Such BSDs do typically occur in the context of BSD-driven video adaptation.

Our contributions in the domain of BSD-driven media resource adaptation have led to the following publications.

1. Davy De Schrijver, Wesley De Neve, Koen De Wolf, Peter Lambert, Davy Van Deursen, Rik Van de Walle. XML-driven Exploitation of Combined Scalability in Scalable H.264/AVC Bitstreams. Submitted to *2007 IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, New Orleans, Louisiana, USA, May 2007.

2. Davy De Schrijver, Wesley De Neve, Koen De Wolf, Robbie De Sutter, Rik Van de Walle. An Optimized MPEG-21 BSDL Framework for the Adaptation of Scalable Bitstreams. Accepted for publication in *Journal of Visual Communication & Image Representation*.

3. Wesley De Neve, Davy De Schrijver, Davy Van Deursen, Rik Van de Walle. XML-driven Bitstream Extraction Along the Temporal Axis

of SMPTE's Video Codec 1. In *Proceedings of the 7th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 233–236, Incheon, Korea, April 2006.

4. Wesley De Neve, Sam Lerouge, Peter Lambert, Rik Van de Walle. A Performance Evaluation of MPEG-21 BSDL in the Context of H.264/AVC. In *Proceedings of SPIE 2004: Signal and Image Processing and Sensors*, Volume 5558, pages 555–566, Denver, Colorado, USA, August 2004.

5. Wesley De Neve, Frederik De Keukelaere, Koen De Wolf, Rik Van de Walle. Applying MPEG-21 BSDL to the JVT H.264/AVC Specification in MPEG-21 Session Mobility Scenarios. In *Proceedings of the 5th International Workshop on Image Analysis for Multimedia Interactive Services*, 4 pages on CD-ROM, Lisboa, Portugal, April 2004.

6. Robbie De Sutter, Sam Lerouge, Wesley De Neve, Peter Lambert, Rik Van de Walle. Advanced Mobile Multimedia Applications: Using MPEG-21 and Time-dependent Metadata. In *Proceedings of SPIE/IT-Com 2003*, Volume 5241, pages 147–156, Orlando, Florida, USA, September 2003.

7. Frederik De Keukelaere, Wesley De Neve, Peter Lambert, Boris Rogge, Rik Van de Walle. MPEG-21 Digital Item Processing Architecture. In *Proceedings of Euromedia 2003*, Eurosis, pages 5–9, Plymouth, UK, April 2003.

Our research in the field of BSD-driven content adaptation also resulted in a few input contributions to MPEG [46,47,49].

Chapter 5

BFlavor: a new bitstream syntax description tool

It is by logic that we prove, but by intuition that we discover.

Jules Henri Poincaré, (1854-1912).

5.1 Introduction

The two bitstream syntax description languages that were reviewed in the previous chapter, in particular MPEG-21 BSDL and XFlavor, can be applied independently for the purpose of BSD-driven media resource adaptation. However, as briefly discussed in the previous chapter, both description tools are characterized by several complementary properties. This observation inspired us to devise a new description tool that harmonizes the two already existing technologies, by combining their strengths and eliminating their weaknesses. In particular, the processing efficiency and expressive power provided by XFlavor on the one hand, and the ability to create high-level BSDs using BSDL on the other hand, were our key motives for the development of a novel BSD-based content adaptation framework.

BFlavor is at the foundation of this harmonized content adaptation framework: it is a new bitstream syntax description tool built on top of XFlavor to efficiently support BSDL features. Hence the name BFlavor, which is a contraction of MPEG-21 BSDL and XFlavor (BSDL + XFlavor)[1]. Starting from a description of the high-level syntax of a media format in BFlavor, it is then possible to automatically generate a format-specific parser that is able to trans-

[1]`http://multimedialab.elis.ugent.be/BFlavor/`.

late the syntax of a compliant bitstream into an XML description. In a next step, this description can be transformed by well-known XML technologies to reflect an appropriate adaptation of the bitstream. This transformed XML description can subsequently be used to create a tailored version of the media resource by relying on a standardized and format-agnostic media processor (i.e., BSDL's BSDtoBin).

This chapter is organized as follows: first, we introduce our harmonized tool for translating the syntax of binary media resources into an XML representation, i.e. BFlavor. Next, a number of experiments are described that were set up to get an estimate of the expressive power and performance of a BFlavor-based framework for video content adaptation. These tests show that current solutions for exposing the syntax of a media resource are outperformed by BFlavor in terms of execution times, memory consumption, and file sizes. Finally, a number of conclusions are drawn.

5.2 The BFlavor tool chain

BFlavor is a new and harmonized description tool for enabling XML-driven adaptation of binary media resources in a format-agnostic way. The design of BFlavor is essentially composed of several extensions to XFlavor, along with a specification of how each XFlavor language construct is mapped to a corresponding BSDL-1 feature.

The next sections provide more details with respect to the overall design of our unified BSD-driven content adaptation chain. An application scenario is provided, as well as a specification of the fundamentals of BFlavor. This specification is not intended to be exhaustive, but is detailed enough to understand the central principles of BFlavor, as well as the motivation for the fundamental design decisions.

5.2.1 Application scenario

This section describes an application scenario that outlines how BFlavor can be used in a more comprehensive framework for the adaptation of binary media resources. BFlavor addresses the same use cases as MPEG-21 BSDL, consequently, the following scenario is adapted from [109]. It involves several content servers streaming live media resources, e.g. *live* news feeds, to end users with different usage environments. These servers are using various coding formats, e.g. MPEG-1 Video, H.264/AVC, et cetera; they rely on the *low-complexity* and *memory-efficient* nature of a BFlavor-driven parser to translate

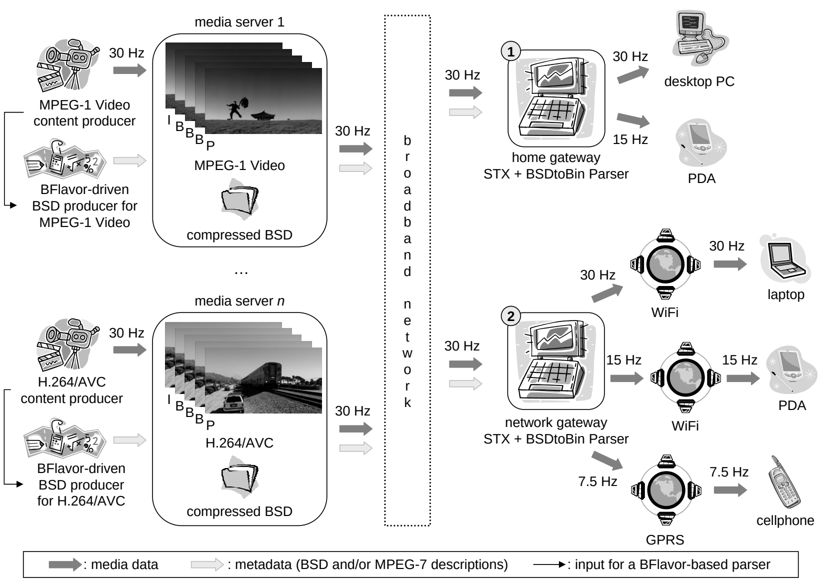

media server 1
30 Hz
MPEG-1 Video content producer
I B B P
MPEG-1 Video
BFlavor-driven BSD producer for MPEG-1 Video
compressed BSD
30 Hz
...
media server n
30 Hz
H.264/AVC content producer
H.264/AVC
BFlavor-driven BSD producer for H.264/AVC
compressed BSD
30 Hz
broadband network
30 Hz
1
home gateway STX + BSDtoBin Parser
30 Hz
desktop PC
15 Hz
PDA
2
network gateway STX + BSDtoBin Parser
30 Hz
WiFi
30 Hz
laptop
15 Hz
WiFi
15 Hz
PDA
7.5 Hz
GPRS
7.5 Hz
cellphone
: media data
: metadata (BSD and/or MPEG-7 descriptions)
: input for a BFlavor-based parser

Figure 5.1: Use case for BFlavor-driven BSD generation in an on-the-fly fashion.

the high-level syntax of a bitstream into an XML description in an on-the-fly fashion.

We assume that technology is available that allows a BSD to be fragmented, compressed, delivered, and processed in pieces, and this in synchrony with the different media streams (video and audio). A number of required technologies have already been standardized. For instance, Binary MPEG format for XML (BiM; [115]) allows the compression of BSDs, while MPEG-21 DIA can be employed for the characterization of a usage environment. Other technologies are the subject of standardization by MPEG at the time of writing, such as MPEG-21 Digital Item Streaming (MPEG-21 DIS; [116]) for BSD fragmentation, synchronized delivery, and piece-wise processing.

The high-level architecture for our use case is visualized in Figure 5.1, showing how BFlavor-driven parsers can complement a BSDL-based adaptive multimedia framework. The actual adaptation of the media content takes place by a format-agnostic content adaptation engine that is deployed on a home gateway and on an intermediate network node connecting two networks with different characteristics. Besides an adaptation decision taking module, the content adaptation engine also comprises a Streaming Transformations for XML (STX) engine for the transformation of BSDs [117], and BSDL's format-neutral BSDtoBin Parser for the construction of a tailored bitstream. The BSD-Link description tool, which is also part of the MPEG-21 DIA standard, can be used to convey possible parameter settings to the different modules composing the content adaptation engine.

The following adaptation scenarios are distinguished:

- *Offline adaptation* - In (1), an agent, acting on behalf of an end user, listens to several non-simultaneous live news feeds of different content providers. Streams of interest are stored on the hard disk of the home gateway of the end user, together with their BSDs and other relevant metadata information (e.g., MPEG-7 metadata, possibly embedded in a BSD by relying on the `bs1:ignore` language construct of MPEG-21 BSDL). The combined use of a STX engine and a BSDtoBin Parser, both running on the home gateway, makes it possible for an application to select the scenes of interest from the different stored news feeds (e.g., according to the end user's preferences), and to customize these data streams (e.g., frame rate reduction) in case the user wants to watch the concatenated scenes on a mobile device with limited resources.

- *On-the-fly adaptation* - In (2), mobile users are connected to the live media servers by a gateway (e.g., at a hot spot on an airport) that interconnects their network [Global Packet Radio Services (GPRS); Wireless Fidelity (WiFi)] with the broadband network of the media servers. On the intermediate network node, an adaptation engine is running. It oversees the usage environments downstream toward the clients and possibly changes the frame rate of incoming media streams in real time to make them suited for playback on the devices of the end users (i.e., to save bandwidth and to reduce the decoding complexity). For example, the adjustment of the frame rate can be realized by dropping sub-sequence layers in case of H.264/AVC bitstreams.

In short, in (1) and (2), the description-driven adaptation is performed on a gateway in the binary domain and independently from the media resource's coding format, rather than during encoding, or decoding and rendering at the client. Note that this approach only requires the implementation of a single, format-independent adaptation engine. As such, it is an alternative to implementing and maintaining a separate adaptation module for each media format encountered. A more thorough discussion on the technical issues that arise from using BSDs in constrained and streaming environments can be found in [109]. The BFlavor description tool addresses one of these challenges - the efficient and real-time generation of BSDs.

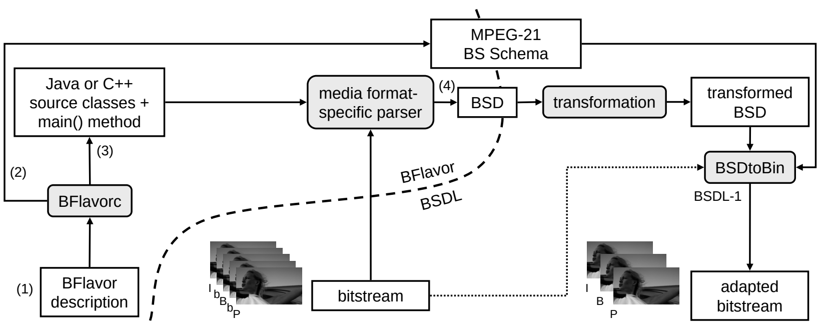


Figure 5.2: The harmonized tool chain, bridging the gap between BSDL and XFlavor.

5.2.2 Harmonized adaptation architecture

BFlavor allows the automatic generation of a set of source classes for a media format-specific parser, as well as the automatic generation of a BS Schema compliant with MPEG-21 BSDL. The intended content adaptation architecture, bridging the gap between XFlavor and BSDL, is depicted in Figure 5.2. Explanatory notes for this figure are provided below:

1. the syntax of a particular media format is described using BFlavor;
2. generation of an MPEG-21 BS Schema by BFlavorc (the modified Flavorc translator);
3. generation of a set of source classes by the BFlavorc translator, constituting the core of a parser able to produce XML output that is in line with the BS Schema generated in step (2);
4. further processing of the XML output of the BFlavor-based parser by a BSD transformation engine and a standardized BSDtoBin Parser, using the automatically created BS Schema for binarization purposes.

The XML output of the BFlavor-based parser is equivalent to that produced by BintoBSD, and hence processable by BSDtoBin. As such, the use of BFlavor-driven parsers, which are format-specific but generated automatically by a format-independent process, is an alternative to the use of a format-neutral BintoBSD Parser. This is an important consideration. The current reference implementation of BintoBSD is characterized by an inefficient behavior in terms of processing speed and memory consumption. These performance issues were already briefly demonstrated in the previous chapter and will be discussed more profoundly in Section 5.3.

Further, a BFlavor-based parser does not need to have a BS Schema at its disposal to generate a BSD. This is due to the format-specific nature of the parser. A BS Schema is only required by BSDtoBin for the construction of an adapted bitstream. Consequently, this allows the automatic creation of BS Schemata that contain a minimal amount of information such that a BSDtoBin Parser can convert each element value in a BSD to a bit-level representation. Such functionality can already be provided by an XML Schema using BSDL-1 datatypes, as BSDL-2 is specific for BintoBSD and not relevant for BSDtoBin. Thus, BSDtoBin may still be used for generating an adapted bitstream without requiring BFlavor to support BSDL-2.

Finally, in order to extend a BintoBSD and BSDtoBin Parser with support for a new media format, for instance, in a distributed adaptive multimedia framework [109], it is only required to send a BS Schema to the network nodes that contain these parsers. However, when such a framework involves BFlavor-based technology for efficient BSD generation, security issues may have to be taken into account as a BFlavor-driven parser is an executable program.

5.2.3 Definition of BFlavor on top of XFlavor

The definition of BFlavor is outlined in the next paragraphs. Actual examples are provided wherever possible to illustrate the principles from a practical point of view. The syntax structures used for the development of BFlavor are the ones relevant to adaptation (i.e., syntax structures up to and including picture or slice headers) of bitstreams compliant with MPEG-1 Video, H.263+, H.264/AVC, and VC-1. The application of BFlavor to MPEG-1 Video and H.264/AVC is extensively discussed in Section 5.3.

Mapping of XFlavor to MPEG-21 BSDL

This section discusses the translation of the most important language constructs of XFlavor into corresponding language features of BSDL. The term “important” refers to the language constructs of XFlavor that were necessary to describe the high-level syntax of the aforementioned coding formats.

Variables. Parsable variables are at the core of XFlavor’s design. As shown in Listing 5.1, these variables include a parse length specification immediately after their type declaration. Because their value is fetched from the bitstream itself, it are parsable variables that define the bitstream syntax. Consequently, parsable variables are mapped by the BFlavorc translator to `xsd:element` language constructs in BSDL.

Listing 5.1: Syntax fragment in BFlavor, containing parsable and non-parsable variables, an operator, and a class.

```
class Example1 {
  // totbits is a non-parsable variable.
  int totbits = numbits();
  if ( totbits > 4 ) {
    // e1 is a parsable variable.
    bit(2) e1;
    bit(e1 + 2) e2;
    Example2 ex2;
  }
}
```

Listing 5.2: BS Schema and BSD fragment, corresponding with the syntax fragment in BFlavor shown in Listing 5.1.

```
<!-- BS Schema fragment, generated by BFlavorc. -->
<xsd:simpleType name="unionType">
  <xsd:union/>
</xsd:simpleType>
<xsd:complexType name="Example1">
  <xsd:sequence>
    <xsd:sequence minOccurs="0">
      <xsd:element name="e1" type="b2"/>
      <xsd:element name="e2" type="unionType"/>
      <xsd:element name="ex2" type="Example2"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="ex1" type="Example1"/>

<!-- BSD fragment, generated by a BFlavor-based parser. -->
<ex1>
  <e1>3</e1>
  <e2 xsi:type="b5">14</e2>
  <ex2><!-- ... --></ex2>
</ex1>
```

XFlavor also has non-parsable variables at its disposal. These variables can be compared to regular variables that are used in traditional programming languages. They do not obtain their value from the bitstream parsed as they are typically used in the context of intermediate computations. As such, non-parsable variables provide similar functionality to BSDL-2 variables, which take the form of XPath expressions. For example, the number of occurrences of a particle (`bs2:nOccurs` attribute), the conditional occurrence of a particle (the `bs2:if` and `bs2:ifUnion` attribute), or the length in bytes of a particle (the `bs2:length` facet). Non-parsable variables are not taken into account by BFlavorc. Indeed, as discussed in the introduction of this section, the automatically generated BS Schema does not need to support BSDL-2: the automatically generated BS Schema is only used for binarization purposes by BSDtoBin, and not for guiding BintoBSD's bitstream parsing process.

Listing 5.1 and Listing 5.2 illustrate how a syntax fragment in BFlavor can be translated into a corresponding BS Schema fragment. A known BSDL-1 datatype is used when it is possible to determine the length of a parsable variable during the conversion. Otherwise, the type of the parsable variable is fixed as an empty `xsd:union` in the BS Schema, meaning that its length has to be explicitly embedded in the instance document in terms of a number of bits by using the `xsi:type` attribute during the BSD generation process. This approach is inspired by the way BSDL's `xsd:union`/`bs2:ifUnion` construct is instantiated in a BSD.

Operators. Operators are built-in functions that facilitate certain frequently appearing data manipulations, such as `lengthof()`, `isidof()`, `skipbits()`, `nextcode()`, `numbits()`, and `nextbits()`. These operators are the only functions that are available in XFlavor. Similar to non-parsable variables, the different operators are not needed for the BSDtoBin process. As such, they are ignored during BFlavorc's translation step. This is for instance demonstrated in Listing 5.1 and Listing 5.2.

Note that the `nextcode()` operator can be used to create a format-specific bitstream extractor (e.g., a tool that drops all bidirectionally coded pictures in a coded video sequence). However, this approach merges the description generation and adaptation processes, which are functional units that will most often be strictly separated from each other in a BSD-based content adaptation architecture. Indeed, thanks to this separation, a maximum number of adaptation steps can be achieved by only having to generate a BSD once ("describe once, transform many times"). Hence, the use of the `nextcode()` operator is inappropriate in the context of BSD-driven content adaptation.

Classes. XFlavor was designed to be a natural extension of the typing system of object-oriented languages like C++ and Java. This means that the bitstream representation information is placed together with the data declarations in a single place. In C++ and Java, this place is where a class is defined. Hence, all datastructures are declared as classes in XFlavor.

The equivalent of a class in XML Schema is a complex type, while objects are essentially equivalent to elements. Therefore, all classes in BFlavor are mapped to complex types by BFlavorc. Object declarations are translated into typed element declarations. This is for example shown in Listing 5.1 and Listing 5.2.

Class arguments. As XFlavor classes cannot have constructors, it is necessary to have a mechanism to pass external information to a class. This is accomplished using class arguments. Similar to non-parsable variables and the built-in operators, class arguments are not required for the BSDtoBin process and are hence not included in the BSDL-1 schema that is generated by BFlavorc. This is illustrated in Listing 5.3 and Listing 5.4.

Expected values. XFlavor has the ability to specify expected values for parsable variables (for example, for bitstream validation). In BFlavor, a range of expected values is only validated during BSD generation, but not during the construction of a bitstream (due to the lack of support for ranges by the `xsd:fixed` attribute in BSDL). This is shown in Listing 5.3 and Listing 5.4.

Listing 5.3: Syntax fragment in BFlavor, containing the `nextbits()` operator, `if`/`while` statements, class arguments, a parsable variable with an expected value, and a one-dimensional array.

```
// arg1 and arg2 are class arguments.
class Example1 (int arg1, int arg2) {
  while ( nextbits(8) == 0xFF )
    bit(8) ff_byte = 0xFF;
  if ( nextbits(1) == 1 )
    bit(1) flag;
  if ( arg1 == 2 )
    bit(2) e1;
  if ( arg2 > 0 )
    bit(2) e2;
}
class Example2 {
  bit(5) e3;
  if ( e3 < 2 ) {
    bit(1) e4;
    Example1 example1 (e3, e4);
  }
  bit(3) e5 [5];
  while ( nextbits(16) == 1 ) {
    bit(2) e6;
    bit(2) e7;
  }
}
```

Listing 5.4: BS Schema fragment, which is the result of the translation of the syntax fragment in BFlavor shown in Listing 5.3.

```
<xsd:complexType name="Example1">
  <xsd:sequence>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="ff_byte" type="b8" fixed="255"/>
    </xsd:sequence>
    <xsd:sequence minOccurs="0">
      <xsd:element name="flag" type="b1"/>
    </xsd:sequence>
    <xsd:sequence minOccurs="0">
      <xsd:element name="e1" type="b2"/>
    </xsd:sequence>
    <xsd:sequence minOccurs="0">
      <xsd:element name="e2" type="b2"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Example2">
  <xsd:sequence>
    <xsd:element name="e3" type="b5"/>
    <xsd:sequence minOccurs="0">
      <xsd:element name="e4" type="b1"/>
      <xsd:element name="example1" type="Example1"/>
    </xsd:sequence>
    <xsd:element name="e5" type="b3"
         maxOccurs="unbounded"/>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="e6" type="b2"/>
      <xsd:element name="e7" type="b2"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
```

Flow control. Listing 5.3 and Listing 5.4 illustrate the straightforward translation of `while` and `if` language constructs in BFlavor to corresponding language features in BSDL. The translation of a `for` loop can be found in the respective BSDs in Appendix A. Because the flow control statements are not needed for BSDtoBin, they are not taken into account during the conversion step from BFlavor to BSDL (e.g., an `if` language construct in BFlavor is not translated into a BS Schema construct using `bs2:if`).

Map declarations. A `map` is used in XFlavor to define constant- or variable-length mappings between bitstream values and object variables. As such, a `map` can be used to implement simple Variable-Length Coding tables (VLC tables), which are often employed for the representation of low-level syntactical units (motion vector differences, transform coefficients, and so on).

Listing 5.5: A `map` in BFlavor, describing the `ptype` syntax element of VC-1.

```
map ptype_map(int) {
        0b0,      0,  // P Picture
        0b10,     1,  // I Picture
        0b110,    2,  // B Picture
        0b1110,  14,  // BI Picture
        0b1111,  15   // Skipped Picture
}
class Picture {
        int(ptype_map) ptype;
}
```

Listing 5.6: BS Schema and BSD fragment, corresponding to the syntax fragment in BFlavor shown in Listing 5.5.

```
<!-- A BS Schema fragment, generated by BFlavorc. -->
<xsd:simpleType name="unionType">
  <xsd:union/>
</xsd:simpleType>
<xsd:complexType name="ptype_map">
  <xsd:sequence>
    <xsd:element name="code" type="unionType"/>
    <xsd:element name="value">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:integer">
            <xsd:attribute ref="bs1:ignore"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<!-- Declaration of a Picture. -->
<xsd:complexType name="Picture">
  <xsd:sequence>
```

```
    <xsd:element name="ptype" type="ptype_map"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="picture" type="Picture"/>

<!-- A sample BSD, generated by a BFlavor-based parser. -->
<picture>
  <ptype>
    <!-- Bitstream value. -->
    <code xsi:type="b3">6</code>
    <!-- Decoded value. -->
    <value bs1:ignore="true">2</value>
  </ptype>
<picture>
```

The translation of a map into BSDL is illustrated in Listing 5.5 and Listing 5.6. When a map declaration is used to describe the layout of a certain syntax element, the bitstream value is usually different from the real or decoded value. Therefore, a map declaration is translated into an element containing two children: an element with the name code and an element with the name value. The code element contains the bitstream value, as well as an xsi:type attribute carrying a length specification. Thus, the code element is used by a BSDtoBin Parser to write a value with the correct length to the bitstream. On the other hand, the value element, containing the decoded value, is ignored by a BSDtoBin Parser as it contains a bs1:ignore attribute with the value true. The value of this element is typically used to steer a BSD transformation engine, which often operates at a semantic level. A similar approach is applied in the BSDs generated by XFlavor, albeit that the length specification is initialized to zero in case of the value element (to make sure that this value is not written to the bitstream by the Bitgen tool).

Note that our approach requires that any transformation tool has to reimplement the mapping when the value of the code element is updated. This means that our current solution duplicates the mapping which already exists in BFlavor, and it also implies that the transformation process must handle part of the bitstream parsing process. An alternative is to have BFlavorc output a class file for the map, after which the BS Schema could refer to this class file using bs0:implementation. As such, the transformation tool only has to deal with the decoded value. This solution has not been implemented in the present version of BFlavorc as an update step was not necessary for the elements described by a map[2].

[2]Due to the high-level nature of our BSDs, a map was required only for a small number of syntax elements in VC-1 (e.g., the ptype syntax element).

Extensions to XFlavor

BFlavor defines a number of extensions to XFlavor by means of three new verbatim codes, two new built-in datatypes, a new built-in operator, and a new built-in base class. These extensions to XFlavor provide functionality that mainly corresponds to the advantages of BSDL identified in Chapter 4 (e.g., the creation of high-level and compact BSDs).

The `root`, `targetns`, and `ns` verbatim codes. XFlavor's verbatim code mechanism allows a developer to insert C++ or Java code in the syntax description of a certain media format. These user code segments are not processed by the Flavorc translator, but are copied verbatim to the C++ or Java output files (which are subsequently provided to a C++ or Java compiler). In BFlavor, this extension mechanism is used to introduce three new verbatim codes. Their use is illustrated in Listing A.3 in Appendix A:

- The `root` verbatim code signals the class to start the parsing with, which makes it possible to automatically create a `main()` function for the parser that initiates the BSD generation process. This is not the case for the original Flavorc translator: it is up to the developer to write a `main()` function with semantics that are dependent on the application targeted (as depicted in Figure 4.3).
- The `targetns` and `ns` verbatim codes convey the information with respect to the different namespaces that are to be used in the intended BS Schemata and BSDs.

The `hexBinary` and `byteRange` datatypes. BFlavor introduces two additional datatypes in XFlavor, in particular the `hexBinary` and `byteRange` datatypes. Their use is illustrated in Listing 5.7 and Listing 5.8.

Listing 5.7: The use of the `hexBinary` and `byteRange` datatypes in BFlavor.

```
// Use of the new hexBinary datatype in BFlavor. The length
// is specified in terms of a number of bytes.
hexBinary(3) startCode;

// Use of the new byteRange datatype in BFlavor.

// Parsing based on the use of start or end codes.
byteRange(3) payload1 = 0x000001 .. 0x000003;

// Parsing based on the use of length fields. The length is
// specified in terms of a number of bytes.
int length = 1500;
byteRange(length) payload2;
```

Listing 5.8: Translation of the `hexBinary` and `byteRange` datatypes in BFlavor to BSDL.

```
<xsd:element name="startCode">
  <xsd:simpleType>
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="3"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="payload1">
  <xsd:simpleType>
    <xsd:restriction base="bs1:byteRange"/>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="payload2">
  <xsd:simpleType>
    <xsd:restriction base="bs1:byteRange"/>
  </xsd:simpleType>
</xsd:element>
```

- The `hexBinary` datatype is typically used for describing start or end codes. A variable of this datatype is translated into a `simpleType` definition, specifying an appropriate restriction to XML Schema's `xsd:-hexBinary` datatype.

- The `byteRange` datatype is used for referring to a particular bitstream segment, addressing the primary disadvantage of XFlavor - this is, the bad granularity of its BSDs. In other words, thanks to the `byteRange` datatype, BFlavor allows the generation of high-level descriptions without losing bitstream data, and hence the creation of descriptions that are much more compact. The datatype `byteRange` is translated into a `simpleType` definition, specifying an appropriate restriction to BSDL's `bs1:byteRange` datatype.

The `align()` operator. Byte-alignment is achieved in BFlavor using the new built-in function `align()`. It is the counterpart of the non-normative `bs0:fillByte` language construct in BSDL. The use of `align()` is shown in Listing A.3.

The `Encoded` **base class.** The class `Encoded` is a simple built-in base class from which other classes may be derived. It specifies a well-defined contract such that BFlavor is able to support an appropriate mapping to BSDL's non-normative `bs0:implementation` extension mechanism.

Listing A.3 and Listing A.4 show how the class `UnsignedExpGolomb` is derived from the base class `Encoded`. `UnsignedExpGolomb` provides the necessary information for a BFlavor-based parser to process a syntax element represented with an Unsigned Exponential Golomb code. The decoded value is subsequently placed by the parser in the BSD. After transformation of the BSD, BSDL's BSDtoBin Parser converts the BSD value to a bitstream representation, again using an Unsigned Exponential Golomb code. This information is provided to BSDtoBin by a procedural object (i.e., a Java class), which is addressed using `bs0:implementation`.

5.3 Performance data

5.3.1 Methodology

The widespread MPEG-1 Video standard and the state-of-the-art H.264/AVC specification were selected as test cases for the validation of the expressive power and performance of a BFlavor-driven tool chain for video adaptation [118], compared to tool chains that are either based on MPEG-21 BSDL or XFlavor. In what follows, a description is given of the common settings used for the two series of conducted experiments: one series of experiments aims at testing the BSD generation performance only (see Section 5.3.2), while a second series of experiments targets the overall performance of BSD-based temporal adaptation in H.264/AVC (see Section 5.3.3).

All experiments were carried out on a PC having an Intel Pentium D 2.8 GHz processor and 1 GB of system memory at its disposal. The operating system used was Windows XP Pro SP2, running Sun Microsystems' Java 2 Runtime Environment (Standard Edition version 1.5.0_05). Version 1.1.3 of the MPEG-21 BSDL reference software was employed, as well as version 5.2.0 of the Flavorc translator for generating Java-based parsers and version 8.6.1 of SAXON for applying XSLT stylesheets to BSDs. A description in BFlavor of the most important syntax structures of MPEG-1 Video and H.264/AVC was used to automatically create a Java-based parser for each of the video coding formats (C++ or C# code could have been generated as well). The maximum heap memory consumption of the different Java-based programs involved was registered by relying on JProfiler 4.0.2.

The MPEG-1 Video bitstreams were created using TMPGEnc 2.5[3], while the H.264/AVC bitstreams were produced by making use of Joint Model (JM) 9.4. Sub-sequence information SEI messages were added to the H.264/AVC bitstreams using the flexibility of the MPEG-21 BSDL framework (metadata injection), since JM 9.4 did not offer support for the creation of such messages. In particular, XSLT stylesheets were used for automatically adding SEI messages to the BSDs for the different H.264/AVC bitstreams, after which BSDtoBin was employed to create new bitstreams containing these messages. These enriched bitstreams could then be used for further processing (BSD generation, BSD transformation, and actual bitstream adaptation). Such a stylesheet is provided in Appendix D (see Listing D.1).

For the first series of experiments, the progressive City test sequence (4CIF; 30 Hz; 4:2:0) was encoded at a bit rate of 1.5 Mbps; for the second series of experiments, the progressive Foreman test sequence (CIF; 300 pictures; 30 Hz; 4:2:0) was used. All H.264/AVC bitstreams are conforming to the Annex B syntax and contain one SPS and one PPS, both stored at the start of the bitstream. The conclusions drawn for MPEG-1 Video also apply to MPEG-2 Video thanks to the similarity in their syntax [79]. All tests were done eleven times, after which an average was taken of the last ten runs to eliminate the start-up latency.

5.3.2 BSD generation performance

This section provides a performance comparison of the various tools for BSD generation - BSDL's BintoBSD Parser, an XFlavor-based parser, and a BFlavor-driven parser. These parsers are able to generate BSDs for bitstreams compliant with MPEG-1 Video or H.264/AVC. Because MPEG-21 DIA does not mandate any specific BS Schema for a particular media format, the level of detail of a description is application-dependent for BSDL and BFlavor.

The syntax of the MPEG-1 Video bitstreams is described up to and including the picture headers, whilst the syntax of the H.264/AVC bitstreams is described up to and including the slice headers. These remarks are also true for XFlavor. However, in contrast to BSDL and BFlavor, the picture and slice data itself are completely embedded in the resulting BSDs since XFlavor does not provide a mechanism for hiding syntactical structures within a referenced byte sequence. The level of detail of the BSDs leaves open applications such as the exploitation of temporal scalability by picture dropping, the exploitation of temporal scalability using placeholder pictures (see Chapter 6), shot or scene selection, correcting wrongly coded syntax elements, et cetera.

[3]Available online: `http://www.tmpg-inc.com/en/index.html`.

Table 5.1: Performance measurements for BSDL, XFlavor, and BFlavor in terms of execution times and memory consumption needed for the generation of a BSD.

		MPEG-21 BSDL			XFlavor				BFlavor		
	BD[a] (s)	ET (s)	GS (PU/s)	MC (MB)	ET (s)	GS (PU/s)	PT (s)	MC (MB)	ET (s)	GS (PU/s)	MC (MB)
MPEG-1 Video	1.7	1.3	39.6	1.6	0.6	87.9	0.1	0.7	0.1	368.2	0.7
	3.3	2.2	44.9	2.0	1.1	94.8	0.3	0.7	0.2	533.3	0.7
	6.6	4.8	41.6	2.5	2.0	98.2	0.2	0.7	0.3	663.3	0.7
	10.0	8.2	36.4	3.4	3.0	99.1	0.5	0.7	0.4	708.2	0.7
	60.0	146.5	12.3	12.0	17.6	102.3	2.8	0.7	2.1	877.9	0.7
	180.0	1075.7	5.0	38.0	52.7	102.4	8.3	0.7	5.9	916.5	0.7
	300.0	2849.9	3.2	64.0	88.7	101.5	13.8	0.7	9.7	923.8	0.7
H.264/-AVC	1.7	8.3	6.1	2.4	0.8	63.8	0.1	0.7	0.1	347.7	0.7
	3.3	21.0	4.8	3.2	1.2	84.9	0.3	0.7	0.2	488.5	0.7
	6.6	79.6	2.5	4.7	2.3	88.4	0.2	0.7	0.3	621.3	0.7
	10.0	164.0	1.8	6.2	3.3	90.1	0.5	0.7	0.4	685.6	0.7
	60.0	4345.6	0.4	28.0	19.4	92.8	2.8	0.7	2.1	877.2	0.7
	180.0	>24h	N/A	N/A	57.3	94.2	7.9	0.7	5.7	943.8	0.7
	300.0	>24h	N/A	N/A	95.9	93.9	12.7	0.7	9.4	954.2	0.7

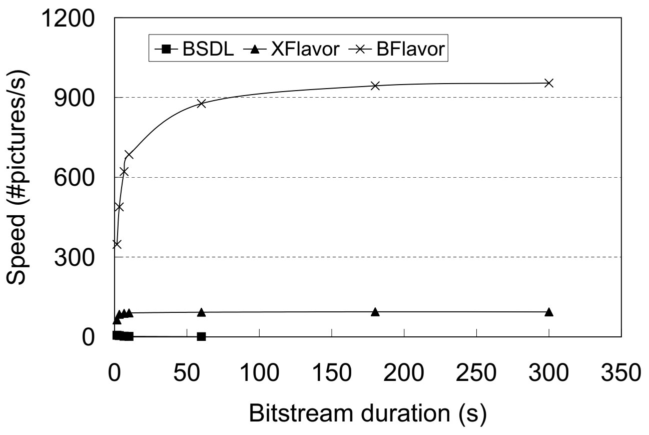
[a]The abbreviations BD, ET, GS, PU, PT, and MC denote Bitstream Duration, Execution Time, Generation Speed, Parse Unit, Parse Time, and Memory Consumption, respectively.

Performance results are reported in Table 5.1, Figure 5.3, and Figure 5.4 by means of computational times, memory consumption, and file sizes. Note the use of logarithmic axes for Figure 5.3(b) and Figure 5.4. As previously discussed, a PU is the fundamental unit of processing for a BSD generator: it denotes a picture in case of MPEG-1 Video and a NAL unit in case of H.264/AVC (each coded picture in our H.264/AVC bitstreams was mapped to one slice).

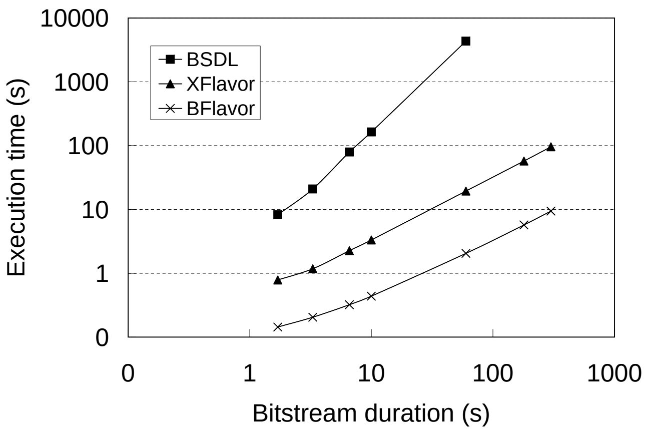
Execution times and memory consumption

A first observation is that both the BFlavor- and XFlavor-based parsers are able to process MPEG-1 Video and H.264/AVC bitstreams at a speed that is at least as fast as the playback speed of the bitstream. For instance, the BFlavor-based tool parses an MPEG-1 Video bitstream, containing 9000 pictures, at a speed of about 923 pictures per second, while an H.264/AVC bitstream, containing 9000 NAL units, is parsed at a speed of about 954 NAL units per second.

A similar real-time behavior can be observed for the XFlavor-based parser, although its parsing speed is lower for both coding formats: the XFlavor-based parser produces verbose BSDs (illustrated by Figure 5.4 for H.264/AVC), re-



(a) Generation speed.



(b) Execution times.

Figure 5.3: BSD generation performance for H.264/AVC.

sulting in a large amount of time needed for writing the BSDs to the hard disk. This can be deduced from the numbers in the columns labeled PT and ET for XFlavor (see Table 5.1). The column PT contains the time needed for executing the parse process only (i.e., BSD generation without writing the resulting description to a storage device).

The impact of the start-up time and the I/O on the speed of the BFlavor-based parser stabilizes (see Figure 5.3(a)) when the H.264/AVC bitstreams have reached a sufficient length (about 1800 pictures, which corresponds to a bitstream duration of 60 s). The constant BSD generation speed of the BFlavor- and XFlavor-based BSD producers also reveals that their processing time is linear in function of the number of parse units, which is an important characteristic in the case of large bitstreams.

From Table 5.1, one can also notice that BSDL's BintoBSD Parser is less efficient than the BFlavor- or XFlavor-based parsers in terms of execution times and memory consumption. The amount of time and memory needed by the BSDL Parser is especially visible in the numbers obtained for H.264/AVC. For instance, the BSDL parser needs about 164 s and 6.2 MB of memory to generate a BSD for an H.264/AVC bitstream with a duration of 10 s, while this parser needs about 8.2 s and 3.4 MB of memory to generate a BSD for an MPEG-1 Video bitstream with the same duration. As shown in Table 5.2, this can be explained by the nature of our BS Schemata, used for discovering the high-level syntax of MPEG-1 Video or H.264/AVC compliant bitstreams.

- The number of syntax elements that is retrieved from an H.264/AVC bitstream is much larger than the amount of syntax elements that is fetched from an MPEG-1 Video bitstream. This can be derived from the number of occurrences of the `xsd:element` and `bs2:nOccurs` language constructs in the respective BS Schemata. Note that the number of retrieved syntax elements is also dependent on the features offered by the coded bitstream. For instance, the BS Schema for H.264/AVC contains support for fetching Video Usability Information (VUI) from an SPS syntax structure. However, the test bitstreams used did not contain these metadata items.

- The BS Schema for H.264/AVC contains a lot of XPath expressions, used as values for the `bs2:nOccurs`, `bs2:if`, and `bs2:ifUnion` attributes. Most of these XPath expressions are necessary to gain access to information that is already retrieved from the bitstream, in particular to information that is stored in the SPS and PPS syntax structures[4]. BintoBSD needs this information to correctly discover the bitstream syntax. This observation is in contrast to the one made for the BS Schema for MPEG-1 Video: it only contains a limited number of straightforward XPath expressions.

The memory consuming behavior of BSDL's BintoBSD Parser is due to the fact that the entire BSD is kept in the system memory. This allows the use of an XPath engine to evaluate a set of arbitrary XPath 1.0 expressions in an at run time fashion, i.e. while parsing the bitstream and progressively generating

[4]The XPath expressions in the BS Schema for H.264/AVC, as used for this series of experiments, take into account the fact that only one SPS and PPS is present in the test bitstreams (see Section 4.4.3 in Chapter 4)

Table 5.2: Number of occurrences of certain language constructs in the MPEG-21 BS Schemata for MPEG-1 Video and H.264/AVC.

language construct	MPEG-1 Video	H.264/AVC (no SEI)	H.264/AVC (with SEI[a])
xsd:element	86	362	209
bs2:nOccurs	0	7	5
bs2:if	4	124	48
bs2:ifUnion	0	29	3
bs2:ifNext	10	12	8
bs2:length	0	0	0
bs2:startCode	8	2	2

[a]SEI messages are used in the second series of experiments.

its BSD. However, this also implies that the internal memory representation of the bitstream syntax description (i.e., the context) increases linearly for every picture (MPEG-1 Video) or NAL unit (H.264/AVC) parsed. The latter observation is not true for the XFlavor-based and BFlavor-driven BSD producers. Their peak memory consumption, used for the temporary storage of the values of parsable and non-parsable variables, is constant (0.7 MB) and independent of the number of units parsed when dealing with MPEG-1 Video and H.264/AVC. Indeed, memory objects, containing information that is no longer necessary to correctly steer the operation of a BFlavor-driven or XFlavor-based parser, are either reused, or either destroyed and subsequently removed from the system memory by the Java garbage collector.

The time consuming behavior of BSDL's BintoBSD Parser, as available in the MPEG-21 reference software package, is mainly caused by the XPath evaluation process.

1. The use of the Xalan XPath engine requires a translation of the internal Document Object Model (DOM) representation of a BSD to a Document Table Model (DTM) representation to allow the evaluation of an arbitrary set of XPath 1.0 expressions.

2. The increasing number of parse units that is stored in the internal DOM or DTM representation results in an increasing number of node tests and predicate evaluations to correctly process an XPath expression. Replacing the Xalan XPath engine by another XPath engine such as Saxon makes it possible to avoid a conversion from a DOM to a DTM representation. However, this will not result in a constant number of operations

(i.e., node tests and predicate evaluations) that needs to be executed during the actual evaluation of an XPath expression as the context keeps growing.

3. The complexity of the XPath expressions, in terms of location steps and predicates used, also influences the amount of time needed by the XPath evaluation process. This is especially true for the XPath expressions that are embedded in the BS Schema for H.264/AVC (e.g., see Listing 4.12).

As such, it is easy to see that the first two observations result in the following behavior: the more units parsed, the larger the context, and hence, the more time needed for evaluating a particular set of XPath expressions in BSDL's BintoBSD Parser. Again, the XFlavor- and BFlavor-driven BSD producers are not characterized by a time consuming behavior during the automatic generation of a BSD. Indeed, these parsers use class parameters, as well as the `[]` indexing and `.` selection operator to gain access to context information.

To summarize, the performance of the reference implementation of BSDL's format-agnostic BintoBSD Parser is infeasible. The processing speed is much slower than real-time and decreases exponentially as the bitstream duration increases. The memory consumption is untenable as well as the BSDL parser keeps the entire BSD in the system memory to support the at-run evaluation of an arbitrary set of XPath 1.0 expressions. The growing context in the BSDL Parser is also recognized as a problem by the developers of the MPEG-21 BSDL specification. For instance, in [109], the authors warn of the potential for memory overflows in the case of large bitstreams or streaming scenarios. However, the increasing size of the internal representation of a BSD not only results in a memory overflow for large bitstreams, it also causes, as shown by our quantitative analysis, a gradual slow-down in the processing speed of the BSDL Parser due to an increase of the time needed for the evaluation of XPath 1.0 expressions.

Several solutions and alternatives can be proposed in order to generate BSDs in a more efficient way.

1. Proprietary (i.e., media format-specific) software can be used for generating BSDs. However, this alternative to the use of BintoBSD is characterized by a lack of genericity.

2. The BSDL specification can be improved to deal with this fundamental problem of BintoBSD, for instance, by introducing context management control operations in the BSDL schema language. These operations can be employed to guarantee a context with a constant size, implying that

constant and faster BSD generation speeds can be achieved. However, this solution requires modifications to be made to the standard and current implementations of BSDL's BintoBSD Parser (including modifications to BS Schemata), as well as an explicit context management by the author of a BS Schema. This solution is discussed in more detail in Appendix C.

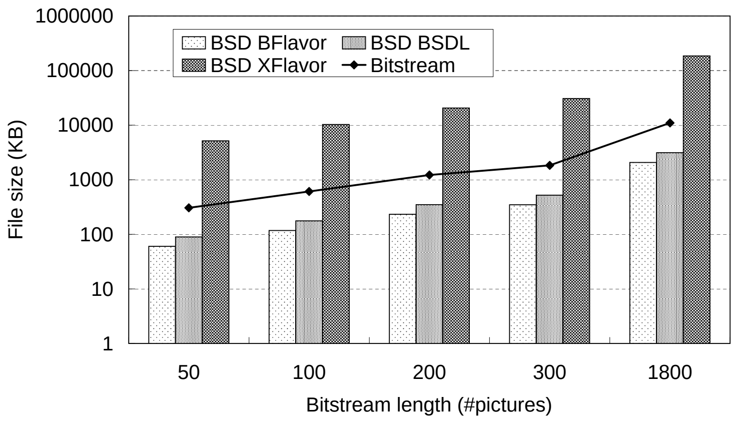
3. Engines can be used that are able to evaluate XPath expressions in a streaming fashion, making it possible to deal with large XML documents. Such streaming XPath engines are already described in the scientific literature [119] [120]. However, most of them are still experimental (e.g., no support for the entire XPath 1.0 specification), or they are not publicly available for experimentation.

On the other hand, as clearly shown by the measurements in Table 5.1, Figure 5.3, and Figure 5.4, BFlavor does offer a working and practical solution for a known challenge [24] [109] [121], i.e. the memory-efficient and real-time generation of BSDL compliant BSDs, by sacrificing a minimal amount of genericity.

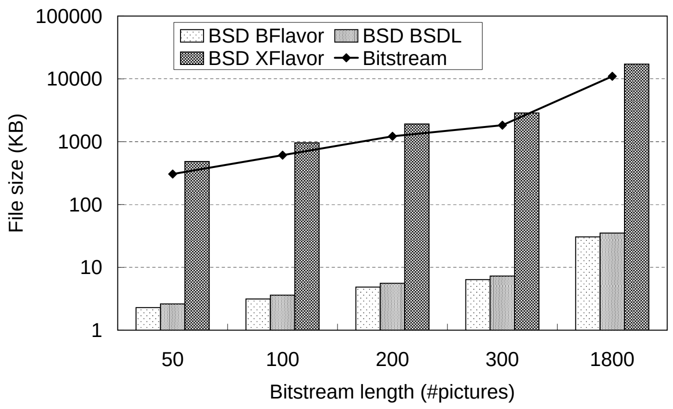
BSD sizes

Figure 5.4 illustrates that the BSDs, generated by BSDL's BintoBSD tool and the BFlavor-driven parser, are compact compared to the BSDs created by the XFlavor-based parser. Indeed, the BSDs as generated by XFlavor are verbose (e.g., a textual BSD size of 181 MB for an H.264/AVC bitstream of 11 MB), because all bitstream data are embedded in the BSD in order not to loose crucial information : every four bytes in the payload are mapped to a 32-bits integer, the value of which is written to the resulting BSD (see Listing A.2 and Listing A.6). As such, the BSDs of BSDL and BFlavor fulfill the goal to act as an additional metadata layer on top of a (scalable) media resource.

Note that the BSDs, produced by BFlavor, are typically smaller than the ones created by BSDL's BintoBSD process, although the same syntax is described. This is due to the fact that BFlavor generates simplified BS Schemata, and hence, also simplified BSDs. Indeed, in contrast to our manually created BS Schemata, supporting BSDL-1 and BSDL-2 and aiming at readability, a BS Schema generated by BFlavorc does not contain, e.g., explicit element declarations to associate a `bs2:if` attribute with. Instead, an `xsd:sequence` statement is used then, which is not instantiated in the output BSD. This is for instance clarified in Listing 5.3 and Listing 5.4 for the automatic translation of the different `if` statements.



(a) Uncompressed BSD sizes.



(b) Compressed BSD sizes.

Figure 5.4: BSD sizes for H.264/AVC.

In Figure 5.4, the sizes of the different encoded bitstreams are given as well. These numbers are rather informative. For instance, in the context of scalable video coding, one of the target applications of BSD-based media content adaptation, bitstreams will typically be encoded at the highest quality possible (in terms of spatial, temporal, and Signal-to-Noise Ratio quality), and hence, at a higher bit rate than the one used in this research (1.5 Mbps due to the use of MPEG-1 Video). As such, a maximum degree of freedom can be achieved when child bitstreams have to be extracted. Nonetheless, it is interesting to see that in the case of BSDL and BFlavor, the ratio, defined as the division of the uncompressed BSD size by the bitstream size, is about 10% for MPEG-1 Video, while this ratio has a value of about 1461% for XFlavor.

Finally, Figure 5.4 also shows that the resulting BSDs can be compressed efficiently when using the default text compression algorithm of WinRAR 3.5. The overhead drops to less than 1% in case of BSDL and BFlavor; it remains about 155% in case of XFlavor. It is expected that the obtained compression ratios are a good indication for those achieved by tools built on top of BiM[5]. At the time this research was performed, the BSDs could not be compressed with the reference software for BiM, due to the lack of support for a number of BSDL-2 language features.

5.3.3 Temporal adaptation performance

In this section, the use of BSDL and BFlavor is discussed for the BSD-driven exploitation of multi-layered temporal scalability in the first version of the H.264/AVC standard. XFlavor is not used in this second series of experiments: as previously shown, its BSDs are too verbose and the language does not explicitly target the adaptation of scalable bitstreams.

Performance results are presented for two use cases: a download-and-play scenario and a simulstore-based streaming scenario. The streaming scenario was outlined in more detail in Section 5.2.1. In both test cases, the goal is to make use of H.264/AVC bitstreams with CIF resolution, having provisions for the exploitation of multi-layered temporal scalability such that three different usage environments can be targeted: a laptop able to process video data at 30 Hz, a portable entertainment device able to process video data at 15 Hz, and a cellphone able to process video data at 7.5 Hz. The generated bitstreams fulfill the requirements as imposed by the sub-sequence coding technique and are compatible with H.264/AVC's widely used Main Profile. Regarding the streaming scenario, three slices per picture are used for improved error robustness while the download-and-play scenario only uses one slice per picture.

Experiments were carried out for H.264/AVC bitstreams having one of the following four coding patterns (the index denotes the number of the sub-sequence layer the picture belongs to): $I_0p_2P_1p_2P_0$, $I_0b_2B_1b_2P_0$, $I_0p_3P_2p_3P_1p_3P_2p_3P_0$, and $I_0b_3B_2b_3B_1b_3B_2b_3P_0$. The $I_0b_2B_1b_2P_0$ coding structure, enabling three-level temporal scalability, is visualized in Figure 3.11 in Chapter 3. Figure 3.9 contains in its turn a visualization of the $I_0b_3B_2b_3B_1b_3B_2b_3P_0$ coding structure.

For bitstreams that do not carry sub-sequence information SEI messages, the syntax elements `frame_num` and `nal_ref_idc` are used to guide the BSD transformation process. This information is available in H.264/AVC's

[5]Besides functionality for lossless compression of XML documents, BiM also offers support for features such as random access and streaming of the binarized documents.

VCL and NAL. For bitstreams containing sub-sequence information SEI messages, the sub-sequence layer identification information (stored in the syntax element `sub_seq_layer_num`) and the value of `nal_unit_type` are used to guide the transformation process (see Listing D.2 in Appendix D). This information only needs to be retrieved from H.264/AVC's NAL.

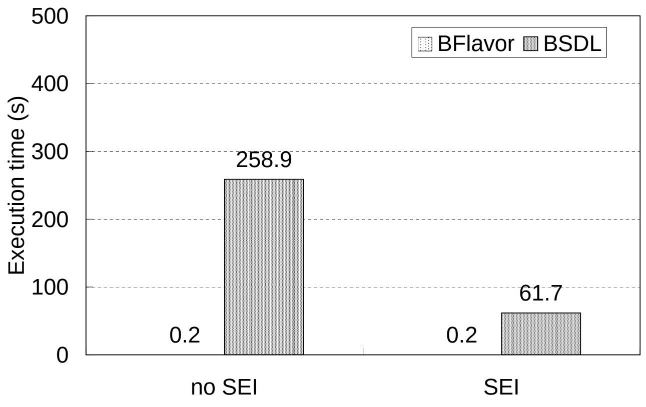
The results in Figure 5.5 and Table 5.3 will be discussed mainly from the most challenging scenario's point of view, i.e. the streaming scenario. This discussion also emphasizes the BSD generation process as BFlavor only affects the first step in a BSD-driven content adaptation chain. Because the values, as obtained for the different metrics, are almost all independent from a particular coding pattern chosen, they were averaged out over the different coding structures used.

Indeed, the coding pattern only has a clear impact on the amount of data dropped: obviously, the bit rate reduction is higher when exploiting temporal scalability in case of a coding structure that embeds P slice coded pictures instead of B slice coded pictures. For example, when downsampling the frame rate of a bitstream from 30 Hz to 7.5 Hz in case of the streaming scenario, the file size drops to 56.9% for the $I_0p_3P_2p_3P_1p_3P_2p_3P_0$ coding structure and to 70.1% for the $I_0b_3B_2b_3B_1b_3B_2b_3P_0$ coding pattern.

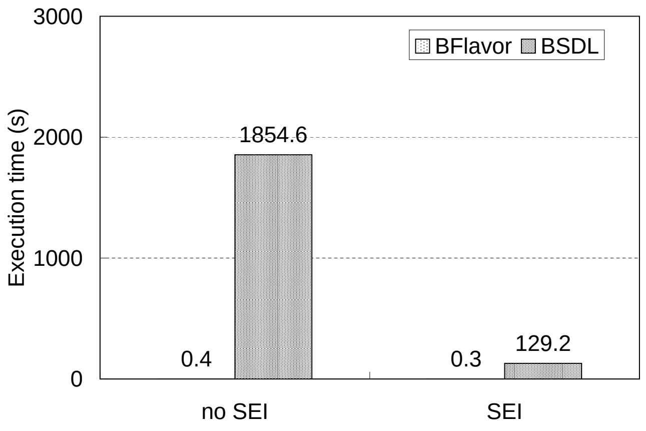
BSD generation without using SEI

The amount of time needed by BSDL's BintoBSD Parser for the generation of a BSD is unacceptably high: 259 s and 1855 s on the average for the download-and-play and streaming scenario, respectively. This can be explained by the fact that H.264/AVC's syntax is described with a rather fine granularity; the resulting BSDs contain information up to and including the slice header syntax structure. As a consequence, a lot of XPath expressions have to be executed to correctly guide BSDL's BintoBSD Parser. Moreover, the detailed analysis of the H.264/AVC bitstreams also results in a quick growth of the context. On the other hand, the BFlavor-based parser creates an equivalent BSD in less than one second.

The BSDs, created by BSDL's BintoBSD Parser, are also more verbose compared to the BSDs produced by our BFlavor-driven parser: for the streaming scenario, the average size of a full BSD is 1263.8 KB for BFlavor and 1958.3 KB for BSDL. As previously explained, this is due to the fact that BFlavor uses simplified BS Schemata and BSDs. Besides the exploitation of temporal scalability by dropping particular sub-sequence layers, the level of detail of the BSDs also enables the exploitation of temporal scalability using placeholder slices (see Chapter 6). To conclude, the BSD granularity represents a



(a) Download-and-play scenario.



(b) Streaming scenario.

Figure 5.5: Execution times for BSD generation.

worst-case scenario for BSD generation in the context of the H.264/AVC video coding format.

BSD generation using SEI

When using sub-sequence information SEI messages for the exploitation of temporal scalability, it is no longer necessary to interpret information from H.264/AVC's VCL; one can entirely rely on the information as available in the NAL. In other words, the sub-sequence information SEI messages assist in abstracting the complexity of the coding format described, facilitating a BS Schema that is much simpler (as indicated by the frequency of certain BSDL

Table 5.3: Summarized performance results for the download-and-play and streaming scenario. BSD generation, transformation, and bitstream reconstruction are expressed in terms of seconds. The sizes of the textual and binarized BSDs are provided in KB.

			frame	BSD gen.		BSD trans.		BSD size		BSDtoBin	
tool	scenario	SEI	rate	avg.	stdev.	avg.	stdev.	text.	bin.	avg.	stdev.
BFlavor	download-and-play	no	30	0.236	0.012	-	-	423.5	8.0	-	-
			15	-	-	0.453	0.002	239.8	5.0	2.895	0.163
			7.5	-	-	0.443	0.006	123.5	3.5	2.241	0.075
		yes	30	0.242	0.008	-	-	402.0	6.0	-	-
			15	-	-	0.485	0.000	210.5	3.0	2.180	0.166
			7.5	-	-	0.484	0.000	126.8	2.0	1.781	0.073
	streaming	no	30	0.356	0.007	-	-	1263.8	17.3	-	-
			15	-	-	0.773	0.006	712.3	10.0	4.155	0.176
			7.5	-	-	0.727	0.009	363.8	6.0	3.003	0.097
		yes	30	0.292	0.010	-	-	637.8	11.5	-	-
			15	-	-	0.635	0.007	331.8	6.0	2.545	0.184
			7.5	-	-	0.613	0.002	171.8	4.0	2.012	0.075
BSDL	download-and-play	no	30	258.878	13.340	-	-	654.9	8.0	-	-
			15	-	-	0.562	0.001	351.8	5.4	3.017	0.162
			7.5	-	-	0.521	0.006	189.5	3.6	2.361	0.083
		yes	30	61.677	0.213	-	-	481.4	5.6	-	-
			15	-	-	0.494	0.004	244.0	4.0	2.651	0.161
			7.5	-	-	0.485	0.001	125.0	2.8	2.273	0.070
	streaming	no	30	1854.571	97.689	-	-	1958.3	18.1	-	-
			15	-	-	0.941	0.009	1047.5	10.6	4.317	0.190
			7.5	-	-	0.818	0.009	560.8	6.3	3.144	0.105
		yes	30	129.210	0.368	-	-	793.8	11.7	-	-
			15	-	-	0.651	0.003	401.0	7.3	2.973	0.186
			7.5	-	-	0.620	0.002	204.0	4.5	2.469	0.072

language constructs in Table 5.2). This can also be derived from the cost needed for generating a BSD: the amount of time has dropped significantly in case of the BSDL Parser (from 1855 s on the average to 129 s on the average for the streaming scenario), mainly because of a smaller context and the fact that less XPath expressions have to be evaluated.

The uncompressed or textual BSDs are also smaller compared to the BSDs generated in the previous approach. However, the former high-level BSDs do no longer allow the exploitation of temporal scalability using placeholder slices as this technique requires the complete parsing of every slice header (to know the boundary between the `slice_header()` and `slice_data()` syntax structures). The granularity of the BSDs represents a best-case scenario for BSD generation for H.264/AVC when aiming at the exploitation of temporal scalability by the removal of particular sub-sequence layers. Nonetheless, the performance of BSDL's BintoBSD process is still unacceptable due to a context of which the growth cannot be controlled.

Finally, the presence of the sub-sequence information SEI messages has little impact on the size of the compressed bitstreams, whilst those content adaptation hints are well suited for enabling fast and intelligent BSD-driven exploitation of temporal scalability in H.264/AVC bitstreams. For example, for the $I_0b_3B_2b_3B_1b_3B_2b_3P_0$ coding pattern in the streaming scenario, the bitstream size without SEI is 332.9 KB while the size of the bitstream enriched with SEI amounts to 335.7 KB (an increase of the file size of about 0.84%).

BSD transformation and bitstream reconstruction

The transformation of all BSDs, using XSLT, can be done in less than one second. Adapted bitstreams can also be generated very quickly, using the BSDtoBin Parser as available in the BSDL reference software. In contrast to BSDL's BintoBSD Parser, this reverse process does not require the evaluation of XPath expressions. Indeed, the BSDtoBin Parser only needs to take care of the selection of appropriate data packets from the original bitstream (by performing look-ups in the transformed BSD) and of the appropriate binarization technique (by doing look-ups in the transformed BSD and/or the BS Schema). Finally, the execution times of the BSD transformation and bitstream reconstruction processes are positively affected by the smaller sizes of the BSDs as produced in the best-case scenario.

5.3.4 Concluding remark

BSDL's BintoBSD and BSDtoBin Parser are characterized by the same asymmetric relation in terms of computational complexity as a typical encoder and decoder pair for the compression of digital video: where the complexity of an encoder is at large determined by the motion estimation process, the complexity of BintoBSD is at large determined by the XPath evaluation process. As such, the computation of a motion vector by an encoder can be compared to the evaluation of an XPath expression by BSDL's BintoBSD Parser. On the other hand, the complexity of a decoder is limited as a motion vector is readily available for the construction of a motion-compensated picture. The same is true for BSDtoBin, where the result of the evaluation of an XPath expression is stored in a BSD, which is readily available for the generation of an adapted bitstream using BSDtoBin.

Exploring the analogy further, the time- and memory-consuming behavior of BSDL's BintoBSD Parser can even be put on par with the behavior of an encoder that would perform motion estimation on an increasing number of reference pictures, and where these pictures are never removed from the encoder's decoded picture buffer.

5.4 Conclusions and original contributions

The initial version of the MPEG-21 BSDL specification is characterized by a fundamental performance problem pertaining to the automatic generation of BSDs for elementary video bitstreams. Indeed, real-life implementations of the format-agnostic BintoBSD process are required to keep the entire BSD in the system memory. This allows the evaluation of a set of arbitrary XPath 1.0 expressions in an at run time fashion, i.e. while parsing the bitstream and progressively generating its BSD. Consequently, this requirement results in an increasing memory usage and a decreasing processing speed for a BintoBSD Parser during the generation of an XML description for the high-level structure of a binary media resource.

Several solutions can be proposed to improve the performance behavior of BSDL's BintoBSD process. In this chapter, we have introduced an alternative to the use of BSDL's format-agnostic BintoBSD Parser, i.e., BFlavor. This new description tool is built on top of XFlavor in order to efficiently support MPEG-21 BSDL features. Its design is essentially composed of several extensions to XFlavor, along with a specification of how each XFlavor language construct is mapped to a corresponding BSDL-1 feature.

BFlavor harmonizes BSDL and XFlavor by combining their strengths and eliminating their weaknesses. More precisely, the processing efficiency and expressive power of XFlavor, together with the ability of BSDL to create high-level BSDs, were our key motives for the development of this new bitstream syntax description tool. As such, the way in which a BFlavor-based BSD generator is created (automatic and programming language-agnostic code generation) can be seen as a trade off between the use of format-agnostic BSD producers on the one hand, and the employment of dedicated and efficient parsers on the other hand. The summary overview in Table 5.4 highlights the major differences and similarities between BSDL, XFlavor, and BFlavor.

To get an estimate of the expressive power and performance of a BFlavor-driven content adaptation chain, several experiments were conducted.

- A first series of experiments targeted a performance analysis of MPEG-21 BSDL, XFlavor, and BFlavor in the context of the automatic generation of BSDs for the MPEG-1 Video and H.264/AVC video coding formats. The BFlavor-based BSD producers achieved similar or significantly better results than their counterparts on all metrics applied (execution times, peak memory consumption, and file sizes).

- A second series of experiments focused on the use of MPEG-21 BSDL and BFlavor for the exploitation of multi-layered temporal scalability in H.264/AVC. Special attention was paid to the combined use of sub-sequences and sub-sequence information SEI messages. These SEI messages allow to abstract the complexity of the coding format described. Also, for this first series of experiments, BFlavor was the only tool to offer an elegant and practical (i.e., real-time) solution for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain.

In short, we have shown in this chapter that BFlavor is an efficient and harmonized description tool for enabling XML-driven adaptation of binary media resources in a format-neutral way.

Finally, we would like to draw the attention of the interested reader to Appendix C and Appendix E.

- Appendix C describes a number of enhancements to the BSDL schema language that allow a BintoBSD Parser to achieve an efficient context management during the generation of a BSD. These extensions were developed by Davy De Schrijver in the context of his research in the domain of BSD-driven content adaptation [32].

- Appendix E contains a brief overview of the second amendment to the MPEG-21 DIA specification, to which Multimedia Lab actively contributed (see for instance [43], [45], and [47]). This amendment to DIA, which is still under development at the time of writing, contains a number of tools that improve the expressive power of BSDL. These tools offer a standardized solution for a number of issues as identified in the previous chapter (e.g., support for Exponential Golomb codes). On the other hand, the amendment to MPEG-21 DIA also contains a number of features that allow more efficient implementations of the BintoBSD process.

The research results that are presented in this chapter are also discussed in the following publications.

1. Davy Van Deursen, Wesley De Neve, Davy De Schrijver, Sarah De Bruyne, Rik Van de Walle. Automatic Generation of generic Bitstream Syntax Descriptions. Submitted to *2007 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (IEEE BMS 2007)*, Orlando, Florida, USA, March 2007.

2. Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Sam Lerouge, Koen De Wolf, Rik Van de Walle. BFlavor: a harmonized approach to media resource adaptation, inspired by MPEG-21 BSDL and XFlavor.

EURASIP Signal Processing - Image Communication, 21(10):862-889, November 2006.

3. Davy De Schrijver, Chris Poppe, Sam Lerouge, Wesley De Neve, Rik Van de Walle. Bitstream Syntax Descriptions for Scalable Video Codecs. *Multimedia Systems Journal*, 11(5):403-421, June 2006.

4. Davy De Schrijver, Wesley De Neve, Koen De Wolf, Stijn Notebaert, Rik Van de Walle. XML-Based Customization Along the Scalability Axes of H.264/AVC Scalable Video Coding. In *Proceedings of 2006 IEEE International Symposium on Circuits and Systems (ISM 2006)*, pages 465–468, Island of Kos, Greece, May 2006.

5. Davy Van Deursen, Davy De Schrijver, Wesley De Neve, Rik Van de Walle. A Real-Time XML-based Adaptation System for Scalable Video Formats. In *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, Volume 4261, pages 343–353, October 2006.

6. Davy Van Deursen, Wesley De Neve, Davy De Schrijver, Rik Van de Walle. BFlavor: an Optimized XML-based Framework for Multimedia Content Customization. In *Proceedings of the 25th Picture Coding Symposium*, 6 pages on CD-ROM, Beijing, China, April 2006.

7. Davy De Schrijver, Wesley De Neve, Koen De Wolf, Rik Van de Walle. Generating MPEG-21 BSDL Descriptions Using Context-related Attributes. In *Proceedings of the 7th IEEE International Symposium on Multimedia*, pages 79–86, Irvine, California, USA, December 2005.

8. Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Koen De Wolf, Rik Van de Walle. Using Bitstream Structure Descriptions for the Exploitation of Multi-layered Temporal Scalability in H.264/AVC's Base Specification. In *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2005*, Volume 3767, pages 641–652, October 2005.

Table 5.4: Overview of the design characteristics of MPEG-21 BSDL, XFlavor, and BFlavor. Note that BSDL, XFlavor, and BFlavor are format-agnostic: their vocabulary is not limited to one particular media format, as is the case for MPML.

Criterion	MPEG-21 BSDL	XFlavor	BFlavor
C1. Language			
Developers	Philips Research	Columbia University	Ghent University
Foundation	W3C XML Schema (restrictions, extensions)	C++/Java (restrictions, extensions)	XFlavor (restrictions, extensions)
Community	metadata	developers	metadata and developers
Format-agnostic	yes	yes	yes
Purpose	to abstract media content adaptation	to abstract bitstream parsing	to abstract media content adaptation
Schema-dependent	yes (e.g., bitstream reconstruction)	no (only validation)	yes (e.g., bitstream reconstruction)
Flow control	BSDL-specific attributes (`bs2:if`, `bs2:ifNext`, `bs2:nOccurs`, `bs2:ifUnion`) and an XML Schema element (`xsd:choice`)	C++/Java-based flow control [`if-else`, (`do-`)`while`, `for`, `switch-case`]	C++/Java-based flow control [`if-else`, (`do-`)`while`, `for`, `switch-case`]
Context access	XPath 1.0	class arguments, `[]` operator, `.` operator	class arguments, `[]` operator, `.` operator

Criterion	MPEG-21 BSDL	XFlavor	BFlavor
Variables	BSDL-2 variables	parsable variables, non-parsable variables	parsable variables, non-parsable variables
Variable-length coding (Exp-Golomb, CAVLC, CABAC)	weak	strong	strong
Bitstream validation	possible (`xsd:fixed`)	possible (= operator)	possible (= operator)
C2. BSD			
Structuring	XML	XML	XML
Granularity	high-level	low-level	high-level
C3. BSD generation			
Tool used	BintoBSD Parser v1.1.3	Java-based parser generated by Flavorc v5.2.0	Java-based parser generated by a modified version of Flavorc v5.2.0
Processing speed	slow	fast	fast
Memory consumption	increasing (DOM, DTM)	low and constant	low and constant
C4. Customized bit-stream generation			
Tool used	BSDtoBin Parser v1.1.3	Bitgen v5.2	BSDtoBin Parser v1.1.3
Processing speed	fast	fast	fast
Memory consumption	low (SAX)	low	low (SAX)

Chapter 6

Enhanced BSD-driven adaptation

We adore chaos because we love to produce order.

M. C. Escher (1898-1972).

6.1 Introduction

In the context of digital video coding, it is important to separate the concept of what is encoded in an elementary video bitstream, which is essentially a compact set of instructions to tell a decoder how to decode the video data, from the concept of what is the decision-making process of an encoder. The latter process is not described in a video coding standard, since it is not relevant to achieving interoperability. Consequently, an encoder has a large amount of freedom about how to decide what to tell a decoder to do.

In this chapter, we show how the aforementioned freedom can also be exploited by a content adaptation engine to offer a solution for a number of issues that may occur after the BSD-driven adaptation of elementary video bitstreams in the coded domain. These issues, such as synchronization and conformance problems, may prevent the correct processing of an adapted bitstream by a multiplexer, which is logic responsible for the synchronization and optimized storage of multiple bitstreams in a file container. For instance, after the disposal of particular pictures in certain classes of elementary video bitstreams (see below), a multiplexer might synchronize the remaining pictures with the wrong audio samples when no precautions are taken. Obviously, this behavior cannot be tolerated in an adaptive multimedia environment.

This chapter outlines two methods that can be considered improvements of already existing approaches for BSD-based content adaptation, targeting the exploitation of temporal and ROI scalability. Our enhanced techniques allow to conceal a number of unwanted side-effects for decoders and multiplexers. These side-effects stem from a (BSD-driven) content adaptation step in the compressed domain. Both approaches are studied in more detail in the context of the H.264/AVC standard. The enhanced exploitation of temporal scalability is briefly discussed for a number of other video coding formats as well.

6.2 Enhanced exploitation of temporal scalability

The traditional view of temporal scalability is to remove certain coded pictures from a bitstream while still obtaining a decodable remaining sequence of pictures. This approach is typically applied when using BSD-based bitstream thinning, after which the remaining pictures often have to be synchronized by a multiplexer with the samples of a corresponding audio stream.

Functionality for synchronization purposes is for instance needed in the context of a digital media archive. This archive may contain fully scalable media resources of different media types, allowing to minimize the storage requirements on the one hand (no simulstore) and to take into account the characteristics of different usage environments on the other hand. BSD-driven content adaptation is employed to deal with the different scalable media formats, used to represent still images (e.g. JPEG 2000; [107]), audio resources (e.g., MPEG-4 BSAC; [109]), and video resources (e.g., MPEG-4 SVC; [35]). After an appropriate adaptation of the different types of media resources, they are assembled and stored by a multiplexer in a single data container, which is subsequently delivered to the client who queried the media archive.

However, after the BSD-driven disposal of certain pictures in an elementary video bitstream, it is sometimes impossible for a multiplexer to (re)synchronize the remaining pictures with the samples of a corresponding audio stream. In particular, BSD-based content adaptation may for example lead to synchronization problems for the following two types of elementary bitstreams: video bitstreams that are characterized by a varying coding pattern and a fixed picture rate, as well as video bitstreams that rely on the use of relative per-picture timestamps (e.g., to support a variable picture rate).

For such bitstreams, the BSD-driven exploitation of temporal scalability, for example implemented by dropping B pictures, will respectively result in bitstreams with a variable picture rate and in bitstreams with wrongly timed pictures. Consequently, for a multiplexer, it is impossible to process the resulting adapted bitstreams in a correct manner, due to missing or incorrect timing

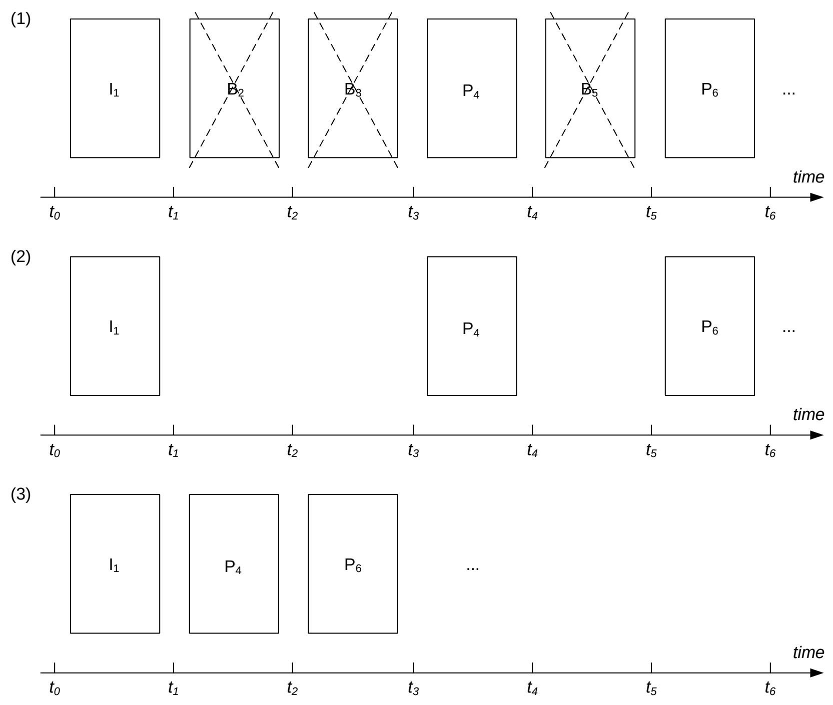


Figure 6.1: Traditional view of temporal scalability.

information. This situation is for instance shown in Figure 6.1. Explanatory notes for this figure are provided below.

1. B pictures are eliminated in an elementary video bitstream that is characterized by a fixed picture rate and a varying coding pattern. The bitstream is coded using an irregular coding structure for the purpose of coding efficiency (e.g., an intra coded picture only occurs at the start of a shot). This bitstream does not convey per-picture timing information.

2. The disposal of B pictures in the elementary video bitstream leads to the creation of gaps in the time line of this adapted bitstream.

3. The gaps in the time line cannot be detected by a multiplexer. As such, the multiplexer incorrectly times the remaining pictures, resulting in an increased playback rate (i.e., the remaining pictures are shown too fast). Consequently, the increased playback speed leads to an increasing time shift (drift) between the playback of the elementary video bitstream and the corresponding audio bitstream.

For elementary bitstreams that are characterized by a regular coding pattern and a fixed picture rate (e.g., IbbPbb...), as well as for bitstreams that rely on the use of absolute per-picture timestamps, the BSD-driven exploitation of temporal scalability usually produces bitstreams that can be processed by a multiplexer in a correct way. Indeed, after the temporal adaptation step, correct timing information is still available, taking the form of a lower but fixed picture rate for the first type of bitstreams, and taking the form of absolute timestamps for the second category of bitstreams[1].

With respect to elementary bitstreams that are either characterized by a varying coding pattern and a fixed picture rate (e.g., video bitstreams stored on commercial movie DVDs), or that rely on the use of relative timestamps, we propose to implement the BSD-driven exploitation of temporal scalability by replacing coded pictures with placeholder pictures. The definition of placeholder pictures, which act as special instructions that can be sent to a decoder, is outlined in the next section from a format-agnostic point of view.

6.2.1 Placeholder pictures

A placeholder or dummy picture is defined as a picture that is identical to a particular reference picture, or that is constructed by relying on a well-defined interpolation process between different reference pictures (e.g., using a weighted average between the co-located pixels in the different reference pictures; see also Section 6.2.3). This means that only a limited amount of information has to be stored or transmitted to signal placeholder pictures to a receiving decoder. As such, placeholder pictures can for example be used by encoders for achieving bit rate savings when the content of successive pictures is very similar. Additionally, these pictures can also be used for synchronization purposes, as they typically introduce an additional delay. Hence, placeholder pictures are sometimes referred to as skipped or delay pictures as well.

In this research, placeholder pictures are used to fill in the gaps that are created in a bitstream due to the disposal of certain pictures. This approach makes it straightforward to maintain synchronization with other media streams in a particular container format: the total number of pictures remains the same after the adaptation step. In other words, the exploitation of temporal scalability, using placeholder pictures, offers a solution for the aforementioned synchronization problems by shielding the systems layer from a number of unwanted side-effects. These side-effects stem from the BSD-driven content adaptation step that is executed at the level of the video coding layer.

[1]Elementary bitstreams often do not carry explicit timing information; this task is typically assigned to the systems layer (e.g., file formats, network protocols), and not to the coding layer.

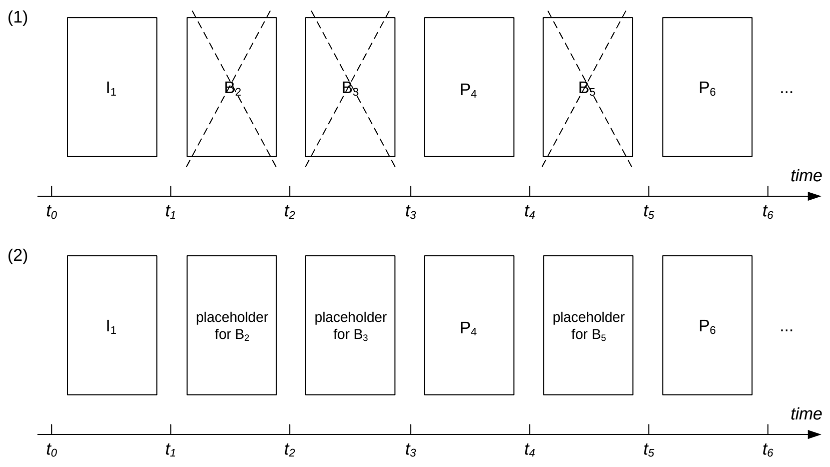


Figure 6.2: Exploitation of temporal scalability by replacing B pictures with placeholder pictures.

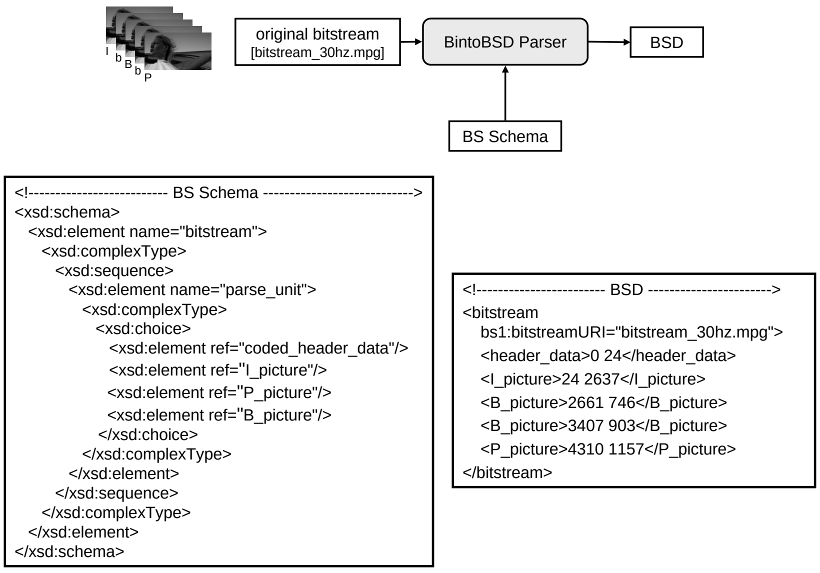
Figure 6.2 illustrates our filling technique: temporal scalability is implemented by replacing B pictures with placeholder pictures. Dependent on the coding format, the placeholder pictures may for instance signal to a decoder to output the previous picture in display order. Finally, from a bitstream perspective, the exploitation of temporal scalability, using placeholder pictures, can be seen as a substitution operation. This is in contrast to the traditional point of view of temporal scalability, which can be considered a removal operation[2].

6.2.2 Placeholder pictures in MPEG-21 BSDL

In order to support the exploitation of temporal scalability using placeholder pictures, additional intelligence needs to be introduced in a BSDL-based content adaptation chain (as visualized in Figure 4.2 in Chapter 4). The required modifications are primarily applied at the level of the BSD transformation (for the purpose of transforming the description of a certain picture into the description of a placeholder picture) and, dependent on the coding format used, at the level of the BS Schema (to allow a correct binarization of placeholder pictures by BSDtoBin). This will be outlined further in this section.

The operational flow regarding BSD-driven exploitation of temporal scalability, using placeholder pictures, is as follows (discussed from a format-independent point of view).

[2]As a side note, the injection of SEI messages into an H.264/AVC bitstream, using MPEG-21 BSDL, can be seen as an addition operation.



```
<!-------------------------- BS Schema ---------------------------->
<xsd:schema>
  <xsd:element name="bitstream">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parse_unit">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="coded_header_data"/>
              <xsd:element ref="I_picture"/>
              <xsd:element ref="P_picture"/>
              <xsd:element ref="B_picture"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
<!------------------------ BSD ----------------------->
<bitstream
   bs1:bitstreamURI="bitstream_30hz.mpg">
   <header_data>0 24</header_data>
   <I_picture>24 2637</I_picture>
   <B_picture>2661 746</B_picture>
   <B_picture>3407 903</B_picture>
   <P_picture>4310 1157</P_picture>
</bitstream>
```

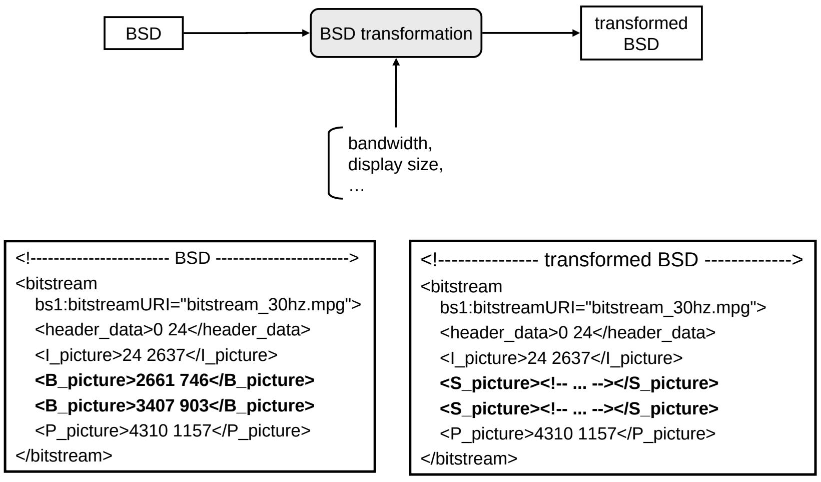
Figure 6.3: BSD generation.

1. Given a bitstream and a BS Schema describing its coding format at a certain granularity, a BSD is created for the bitstream using a format-agnostic BintoBSD Parser. This process is illustrated in Figure 6.3, and is similar to the BSD generation step in a traditional BSDL-based content adaptation chain.

2. A stylesheet (e.g., implemented using XSLT) is employed for iterating through the descriptions of the different parse units in the BSD. Descriptions of certain pictures are transformed into descriptions of placeholder pictures. For instance, descriptions of bidirectionally coded pictures can be replaced by descriptions representing a placeholder or skipped picture. This BSD transformation process is shown in Figure 6.4.

3. An adapted bitstream is generated by BSDL's BSDtoBin process, using the transformed BSD, the original bitstream, and, dependent on the coding format used, an extended BS Schema (see further). This process is visualized in Figure 6.5. The resulting bitstream may subsequently be provided to a multiplexer, which, for example, synchronizes the pictures with the corresponding samples of the original audio stream.

As discussed in more detail in the next section, for a number of video coding formats, the BS Schema has to include an explicit syntax description of a



```
<!----------------------- BSD ---------------------->
<bitstream
   bs1:bitstreamURI="bitstream_30hz.mpg">
   <header_data>0 24</header_data>
   <I_picture>24 2637</I_picture>
   <B_picture>2661 746</B_picture>
   <B_picture>3407 903</B_picture>
   <P_picture>4310 1157</P_picture>
</bitstream>
```

```
<!--------------- transformed BSD ------------->
<bitstream
   bs1:bitstreamURI="bitstream_30hz.mpg">
   <header_data>0 24</header_data>
   <I_picture>24 2637</I_picture>
   <S_picture><!-- ... --></S_picture>
   <S_picture><!-- ... --></S_picture>
   <P_picture>4310 1157</P_picture>
</bitstream>
```

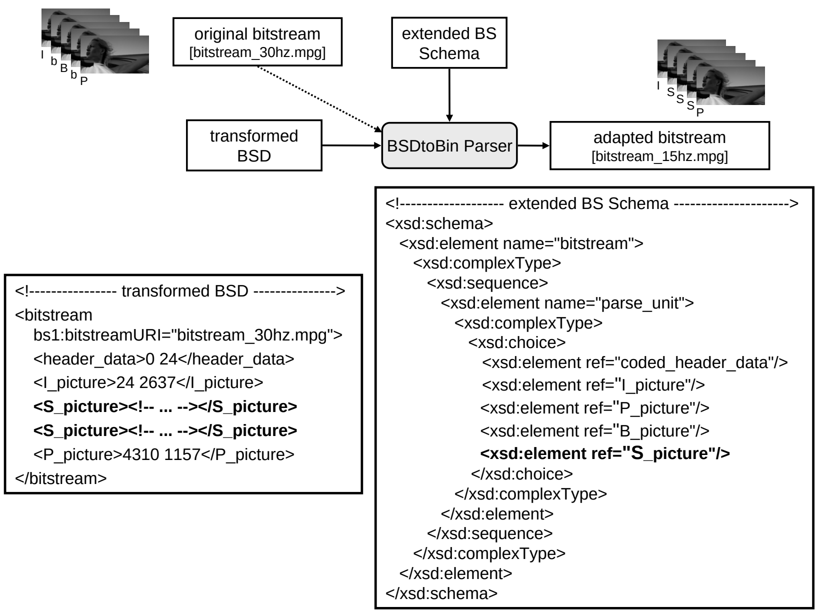
Figure 6.4: BSD transformation: descriptions of B pictures (denoted by `B_picture` in the figure) are translated to descriptions of skipped pictures (denoted by `S_picture` in the figure) in the XML domain.

placeholder picture. This extended BS Schema enables the BSDtoBin process to generate an adapted bitstream containing this type of pictures. The need for an extended BS Schema can be explained by the fact that some video coding formats need a number of low-level syntax elements for the construction of a placeholder picture (e.g., a flag at the level of the macroblock layer that signals to a decoder that all macroblocks of a particular picture or slice are to be considered as skipped).

The part of the BS Schema that describes the syntax of a placeholder picture is only used by a BSDtoBin Parser for binarization purposes; it is not used by the BintoBSD proces as this particular BS Schema fragment is too low-level on the one hand (it contains information that resides at the macroblock layer), and too specific on the other hand (this syntax fragment can only be used for the description of placeholder pictures, and not for the discovery of the syntax of arbitrary coded pictures). Nonetheless, for practical reasons, the extended BS Schema will in most cases also be used by a BintoBSD Parser for the creation of a BSD for a particular video bitstream.

6.2.3 BSD-based construction of placeholder pictures

In this section, the creation of placeholder pictures is outlined in more detail for a number of well-known coding formats: MPEG-1 Video, H.262/MPEG-2 Video, MPEG-4 Visual, VC-1, and H.264/AVC. This discussion is mainly



```
<!---------------- transformed BSD --------------->
<bitstream
   bs1:bitstreamURI="bitstream_30hz.mpg">
   <header_data>0 24</header_data>
   <I_picture>24 2637</I_picture>
   <S_picture><!-- ... --></S_picture>
   <S_picture><!-- ... --></S_picture>
   <P_picture>4310 1157</P_picture>
</bitstream>
```

```
<!------------------- extended BS Schema -------------------->
<xsd:schema>
  <xsd:element name="bitstream">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parse_unit">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="coded_header_data"/>
              <xsd:element ref="I_picture"/>
              <xsd:element ref="P_picture"/>
              <xsd:element ref="B_picture"/>
              <xsd:element ref="S_picture"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 6.5: Adapted bitstream generation by BSDtoBin, eventually using an extended BS Schema for a correct binarization of placeholder pictures.

conducted from a syntax point of view. Special attention is paid to the creation of placeholder pictures in H.264/AVC. The motivation for this decision is twofold. First, H.264/AVC is gaining momentum as the preferred solution for digital video coding, and second, its technical design is the most challenging one for our BSD-based substitution technique.

MPEG-1 Video and H.262/MPEG-2 Video

Signaling a placeholder picture in H.262/MPEG-2 Video can be realized by using so-called pseudo-skipped pictures. A pseudo-skipped picture, introduced by Lee *et al.* in [114] for rate control purposes, is an artificial B picture that consists of multiple slices (one slice for each row). As shown in Figure 6.6, every skipped slice contains two macroblocks that mark the beginning and the end of the slice. In-between macroblocks are automatically considered as skipped by a decoder. The macroblocks at the start and the end of the slice are coded in backward or forward motion compensated mode, as well as with zero-valued motion vectors and quantized transform coefficients.

H.262/MPEG-2 Video can be considered a superset of MPEG-1 Video. As such, the creation of placeholder pictures in MPEG-1 Video is similar: only a

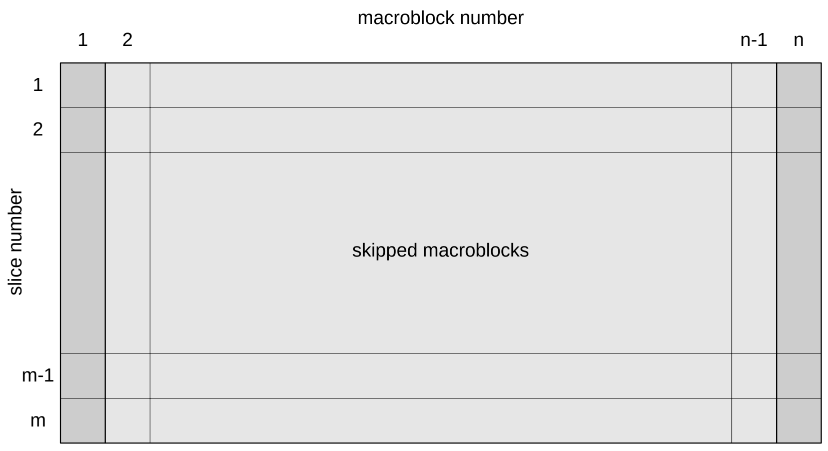


Figure 6.6: Format of a pseudo-skipped picture in H.262/MPEG-2 Video [114].

limited number of minor syntactical differences are to be taken into account at the macroblock layer. Also, the more flexible definition of a slice in MPEG-1 Video allows for a few optimizations since an entire picture can be coded as a single slice. This is not allowed in H.262/MPEG-2 Video: all macroblock rows must start and end with at least one slice (therefore, H.262/MPEG-2 Video cannot be seen as a strict superset of MPEG-1 Video).

A BSD fragment describing a placeholder picture in MPEG-1 Video is depicted in Listing 6.1. This BSD excerpt describes the syntax of a pseudo-skipped picture in MPEG-1 Video, having CIF resolution. Every macroblock row is coded as one slice. Furthermore, each macroblock is coded in backward motion compensated mode, signaling to a decoder that the pseudo-skipped picture is to be replaced by the next (reference) picture in output order. During the BSD transformation step, a stylesheet can be used for transforming the description of a B picture into the description of a placeholder picture. The BSD fragment in Listing 6.1 illustrates a resolution dependency in the number of slices and the macroblock address increment for the last macroblock of a particular slice. This increment is encoded by making use of a variable-length code (the number of macroblocks skipped is equal to the *decoded* value of `macroblock_address_increment` minus one). As such, the number of bits used to represent this increment is dependent on the spatial resolution of the video bitstream (e.g., the bitstream value of 18 for `macroblock_address_increment` in the BSD in Listing 6.1 is decoded to 21, signaling to a compliant decoder that 20 macroblocks are to be considered as skipped - CIF video has 18 rows of 22 macroblocks).

The corresponding BS Schema fragment, which is only used by the BSDtoBin process, can be found in Listing 6.2. This BS Schema excerpt is also resolution-dependent, similar to the BSD fragment as presented in Listing 6.1. Similar descriptions and stylesheets are employed for bitstreams compliant with the H.262/MPEG-2 Video standard.

The resolution dependencies in the BSD and BS Schema excerpts (indicated by a bold font) could for instance be dealt with by making use of parameterized stylesheets and parameterized BS Schemata. Furthermore, the exploitation of temporal scalability in MPEG-1 Video and H.262/MPEG-2 Video, using placeholder pictures, also requires some knowledge, although limited, about a number of low-level syntax structures in both coding formats.

Listing 6.1: BSD fragment describing a pseudo-skipped picture in MPEG-1 Video, having CIF resolution.

```
<pu>
  <pseudo_skipped_picture_352x288>
    <picture_header>
      <picture_start_code>00000100</picture_start_code>
      <temporal_reference>1</temporal_reference>
      <!-- B picture -->
      <picture_coding_type>3</picture_coding_type>
      <vbv_delay>39360</vbv_delay>
      <forward_motion_info>
        <full_pel_forward_vector>0</full_pel_forward_vector>
        <forward_f_code>7</forward_f_code>
      </forward_motion_info>
      <backward_motion_info>
        <full_pel_backward_vector>0</full_pel_backward_vector>
        <backward_f_code>7</backward_f_code>
      </backward_motion_info>
      <extra_bit_picture>0</extra_bit_picture>
      <bit_stuffing>0</bit_stuffing>
    </picture_header>
    <!-- First slice. -->
    <slice_row_352x288>
      <slice_start_code>00000101</slice_start_code>
      <quantiser_scale_code>1</quantiser_scale_code>
      <extra_bit_slice>0</extra_bit_slice>
      <first_macroblock>
        <macroblock_address_increment>
          1
        </macroblock_address_increment>
        <macroblock_type>2</macroblock_type>
        <motion_horizontal_backward_code>
          1
        </motion_horizontal_backward_code>
        <motion_vertical_backward_code>
          1
        </motion_vertical_backward_code>
      </first_macroblock>
      <last_macroblock>
        <macroblock_address_increment>
          18
        </macroblock_address_increment>
        <macroblock_type>2</macroblock_type>
        <motion_horizontal_backward_code>
          1
        </motion_horizontal_backward_code>
        <motion_vertical_backward_code>
          1
        </motion_vertical_backward_code>
      </last_macroblock>
      <bit_stuffing>0</bit_stuffing>
    </slice_row_352x288>
    <!-- ... -->
    <!-- Last slice. -->
    <slice_row_352x288>
      <slice_start_code>00000112</slice_start_code>
      <quantiser_scale_code>1</quantiser_scale_code>
      <extra_bit_slice>0</extra_bit_slice>
      <first_macroblock>
        <macroblock_address_increment>
          1
        </macroblock_address_increment>
        <macroblock_type>2</macroblock_type>
        <motion_horizontal_backward_code>
          1
        </motion_horizontal_backward_code>
        <motion_vertical_backward_code>
          1
        </motion_vertical_backward_code>
      </first_macroblock>
      <last_macroblock>
        <macroblock_address_increment>
          18
        </macroblock_address_increment>
        <macroblock_type>2</macroblock_type>
        <motion_horizontal_backward_code>
          1
        </motion_horizontal_backward_code>
        <motion_vertical_backward_code>
          1
        </motion_vertical_backward_code>
      </last_macroblock>
      <bit_stuffing>0</bit_stuffing>
    </slice_row_352x288>
  </pseudo_skipped_picture_352x288>
</pu>
```

Listing 6.2: BS Schema fragment describing a pseudo-skipped picture in MPEG-1 Video, having CIF resolution.

```
<xsd:element name="pseudo_skipped_picture_352x288">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="mp1:picture_header"/>
      <xsd:element ref="mp1:slice_row_352x288"
                   bs2:nOccurs="18"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="slice_row_352x288">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="slice_start_code"
                   type="mp1:StartCodeType"/>
      <xsd:element name="quantiser_scale_code" type="b5"/>
      <xsd:element name="extra_bit_slice" type="b1"
                   fixed="0"/>
      <xsd:element name="first_macroblock">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="macroblock_address_increment"
                         fixed="1" type="b1"/>
            <!-- macroblock_type is a bitstring equal to 010
                 (bkwd, not-coded). -->
            <xsd:element name="macroblock_type"
                         fixed="2" type="b3"/>
            <xsd:element name="motion_horizontal_backward_code"
                         fixed="1" type="b1"/>
            <xsd:element name="motion_vertical_backward_code"
                         fixed="1" type="b1"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="last_macroblock">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="macroblock_address_increment"
                         fixed="18" type="b10"/>
            <!-- macroblock_type is a bitstring equal to 010
                 (bkwd, not-coded). -->
            <xsd:element name="macroblock_type"
```

```
                        fixed="2" type="b3"/>
            <xsd:element name="motion_horizontal_backward_code"
                        fixed="1" type="b1"/>
            <xsd:element name="motion_vertical_backward_code"
                        fixed="1" type="b1"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <!-- Get back byte-aligned. -->
      <xsd:element name="bit_stuffing"
                  type="mp1:Stuffing" minOccurs="0"/>
      <!-- Look for the start of the next start code. -->
      <xsd:element name="stuffing_byte" type="xsd:unsignedByte"
                  minOccurs="0" maxOccurs="unbounded"
                  bs2:ifNext="00000000"/>
      <xsd:element name="stuffing_byte" type="xsd:unsignedByte"
                  minOccurs="0" maxOccurs="1"
                  bs2:ifNext="00000001"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

MPEG-4 Visual and Video Codec 1

In MPEG-4 Visual, the description of any coded Video Object Plane (VOP), regardless of its coding type, can be transformed into the description of a non-coded VOP (N-VOP), i.e. a placeholder picture, by setting the value of the `vop_coded` flag in the VOP header to zero and by stripping all further information for that particular VOP. This two-step process, which needs to be implemented by the stylesheet that is responsible for the transformation of the BSD, is illustrated in Listing 6.3. An N-VOP instructs a decoder to output the most recent decoded I- or P-VOP for which the `vop_coded` flag is equal to one, preceding the N-VOP in question in output order.

Listing 6.3: Description of a B-VOP and an N-VOP, i.e. a placeholder picture.

```
<!-- Description of a B-VOP. -->
<pu>
  <video_object_plane>
    <vop_start_code>000001B6</vop_start_code>
    <vop_coding_type>2</vop_coding_type>
    <modulo_time_base>0</modulo_time_base>
    <marker_bit>1</marker_bit>
    <vop_time_increment xsi:type="b5">2</vop_time_increment>
    <marker_bit>1</marker_bit>
    <vop_coded>1</vop_coded>
    <vop_stuffing>0</vop_stuffing>
    <vop_payload>47 5746</vop_payload>
  </video_object_plane>
</pu>
<!-- Description of an N-VOP. -->
<pu>
  <video_object_plane>
    <vop_start_code>000001B6</vop_start_code>
    <vop_coding_type>2</vop_coding_type>
    <modulo_time_base>0</modulo_time_base>
    <marker_bit>1</marker_bit>
    <vop_time_increment xsi:type="b5">2</vop_time_increment>
    <marker_bit>1</marker_bit>
    <vop_coded>0</vop_coded>
    <vop_stuffing>0</vop_stuffing>
  </video_object_plane>
</pu>
```

In the context of MPEG-4 Visual, the presence of placeholder pictures can for instance be detected by MP4Box, which is multiplexer software developed by the GNU Project on Advanced Content (GPAC)[3]. In a next step, MP4Box is able to make use of the means of the MP4 file format to create a media container for storing a video bitstream with a variable picture rate (by removing the placeholder pictures from the elementary bitstream and by adjusting the time duration of the remaining pictures). For example, using Figure 6.1, the time duration of picture I_1 would be adjusted from [t_0, t_1] to [t_0, t_3], the time duration of picture P_4 would be adjusted from [t_3, t_4] to [t_3, t_5], et cetera.

SMPTE's VC-1 specification provides an own picture type for signaling a placeholder picture, i.e. a Skipped picture (see Section 4.5 of Chapter 4). Therefore, it can be considered straightforward to translate the description of an arbitrary picture into the description of a Skipped picture (taking into account inter-picture dependencies). A BSD fragment, containing the description of a Skipped picture in VC-1, is shown in Listing 6.4.

Listing 6.4: BSD excerpt describing a Skipped picture in VC-1.

```
<encapsulated_bdu>
  <frame>
    <bdu_start_code>0000010D</bdu_start_code>
    <ptype>
      <Skipped>15</Skipped>
    </ptype>
```

[3]Available online: `http://gpac.sourceforge.net/`.

```
    <skipped_picture_info>
      <if_pulldown_not_0>
        <if_interlace_eq_0_or_psf_eq_1>
          <rptfrm>0</rptfrm>
        </if_interlace_eq_0_or_psf_eq_1>
        <if_interlace_not_0_and_psf_not_1>
          <tff>1</tff>
          <rff>0</rff>
        </if_interlace_not_0_and_psf_not_1>
      </if_pulldown_not_0>
    </skipped_picture_info>
  </frame>
</encapsulated_bdu>
```

Finally, to support the exploitation of temporal scalability in MPEG-4 Visual and VC-1, using placeholder pictures, it is not necessary to embed additional information in the respective BS Schemata. This is due to the inherent support for skipped pictures in both specifications. Furthermore, in contrast to MPEG-1 Video and H.262/MPEG-2 Video, both standards also allow the creation of resolution-independent descriptions of placeholder pictures.

H.264/AVC

In the context of MPEG-1 Video and H.262/MPEG-2 Video, the generic nature of the proposed BSD-driven substitution approach is constrained by a resolution dependency with respect to the creation of placeholder pictures. For H.264/AVC, a number of additional dependencies have to be taken into account when exploiting temporal scalability using placeholder pictures. Indeed, besides the picture resolution and the number of macroblocks used in a slice, it also necessary to be aware of coding parameters such as the entropy coding scheme used, the slice type employed, and whether or not a slice is used as a reference for the reconstruction of other slices. Further, the H.264/AVC syntax also allows the construction of two types of placeholder pictures, each having different semantics for an H.264/AVC decoder.

- A picture consisting of *skipped B slices* signals to an H.264/AVC decoder to output a picture that is constructed by applying a weighted interpolation between a previous and a next picture in output order. More precisely, the interpolated picture is computed based on the relative temporal positions of two (decoded) reference pictures, where the first reference picture is located at index 0 in list 0 and where the second reference

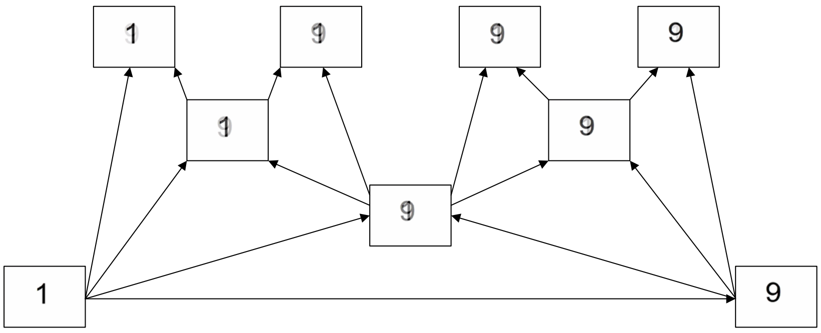


Figure 6.7: Weighted filtering by an H.264/AVC decoder.

picture can be found at index 0 in list 1 in the decoded picture buffer[4].

This weighted filtering process is for instance illustrated in Figure 6.7. It visualizes the output produced by the H.264/AVC reference software decoder for the hierarchical coding structure depicted in Figure 3.9, and in which all B slice coded pictures in the temporal enhancement layers are replaced by pictures consisting of skipped B slices. As such, weighted filtering is applied between the picture labeled "1" and the picture labeled "9" of the base layer.

- A picture consisting of *skipped P slices* instructs an H.264/AVC decoder to output the decoded reference picture that is located at index 0 in list 0 in the decoded picture buffer. This is usually the most recent decoded reference picture that was added to the decoded picture buffer. As such, the pictures in the temporal enhancement layers in Figure 3.9 are replaced by picture "9" in case the B slice coded pictures are substituted for pictures consisting of skipped P slices.

The creation of placeholder slices in H.264/AVC will be exemplified with the translation of the syntax of a non-reference B slice into the syntax of a non-reference skipped P slice (as explained in Chapter 2, slices, and not pictures, are the fundamental unit of processing in H.264/AVC's VCL). This conversion process is clarified in Listing 6.5 and Listing 6.6. The idea is to keep the creation of placeholder slices as simple as possible, minimizing the impact on the different decoding processes, such as the reference picture and display

[4]According to certain rules, decoded reference pictures are organized by an encoder and a decoder in two lists in the decoded picture buffer: list 0 and list 1. P slices are only allowed to refer to pictures in list 0 for their reconstruction, while B slices are allowed to refer to pictures in both lists for their reconstruction.

order management, the operation of the deblocking filter, et cetera. As such, the construction of skipped slices in an H.264/AVC bitstream is realized with a limited number of changes at the level of the different parameter sets and the `slice_header()` syntax structures (see below).

Listing 6.5: BSD fragment for a non-reference B slice in H.264/AVC.

```
<byte_stream_nal_unit>
  <start_code_prefix_one_3bytes>
    000001
  </start_code_prefix_one_3bytes>
  <nal_unit>
    <forbidden_zero_bit>0</forbidden_zero_bit>
    <nal_ref_idc>0</nal_ref_idc>
    <nal_unit_type>1</nal_unit_type>
    <raw_byte_sequence_payload>
      <coded_slice_of_a_non_IDR_picture>
        <slice_layer_without_partitioning_rbsp>
          <slice_header>
            <first_mb_in_slice>466</first_mb_in_slice>
            <slice_type>1</slice_type>
            <pic_parameter_set_id>0</pic_parameter_set_id>
            <frame_num xsi:type="b5">2</frame_num>
            <if_pic_order_cnt_type_eq_0>
              <pic_order_cnt_lsb xsi:type="b7">
                2
              </pic_order_cnt_lsb>
            </if_pic_order_cnt_type_eq_0>
            <if_slice_type_eq_B>
              <direct_spatial_mv_pred_flag>
                1
              </direct_spatial_mv_pred_flag>
            </if_slice_type_eq_B>
            <if_slice_type_eq_P_or_SP_or_B>
              <num_ref_idx_active_override_flag>
                0
              </num_ref_idx_active_override_flag>
            </if_slice_type_eq_P_or_SP_or_B>
            <ref_pic_list_reordering>
              <if_slice_type_not_I_and_not_SI>
                <ref_pic_list_reordering_flag_l0>
                  0
                </ref_pic_list_reordering_flag_l0>
              </if_slice_type_not_I_and_not_SI>
              <if_slice_type_eq_B>
                <ref_pic_list_reordering_flag_l1>
                  0
                </ref_pic_list_reordering_flag_l1>
              </if_slice_type_eq_B>
            </ref_pic_list_reordering>
            <slice_qp_delta>-8</slice_qp_delta>
            <if_deblocking_filter_control_present_flag_not_0>
              <disable_deblocking_filter_idc>
                1
              </disable_deblocking_filter_idc>
            </if_deblocking_filter_control_present_flag_not_0>
          </slice_header>
          <slice_data>
            <bit_stuffing>5</bit_stuffing>
            <slice_payload>20465 15</slice_payload>
          </slice_data>
        </slice_layer_without_partitioning_rbsp>
      </coded_slice_of_a_non_IDR_picture>
    </raw_byte_sequence_payload>
  </nal_unit>
</byte_stream_nal_unit>
```

Listing 6.6: BSD fragment for a skipped P slice in H.264/AVC.

```
<byte_stream_nal_unit>
  <start_code_prefix_one_3bytes>
    000001
  </start_code_prefix_one_3bytes>
  <nal_unit>
    <forbidden_zero_bit>0</forbidden_zero_bit>
    <nal_ref_idc>0</nal_ref_idc>
    <nal_unit_type>1</nal_unit_type>
    <raw_byte_sequence_payload>
      <coded_slice_of_a_skipped_non_IDR_picture>
        <skipped_slice_layer_without_partitioning_rbsp>
          <slice_header>
            <first_mb_in_slice>466</first_mb_in_slice>
            <slice_type>5</slice_type>
            <pic_parameter_set_id>0</pic_parameter_set_id>
            <frame_num xsi:type="b5">2</frame_num>
            <if_pic_order_cnt_type_eq_0>
              <pic_order_cnt_lsb xsi:type="b7">
                2
              </pic_order_cnt_lsb>
            </if_pic_order_cnt_type_eq_0>

            <if_slice_type_eq_P_or_SP_or_B>
              <num_ref_idx_active_override_flag>
                0
              </num_ref_idx_active_override_flag>
            </if_slice_type_eq_P_or_SP_or_B>
            <ref_pic_list_reordering>
```

```
              <if_slice_type_not_I_and_not_SI>
                <ref_pic_list_reordering_flag_l0>
                  0
                </ref_pic_list_reordering_flag_l0>
              </if_slice_type_not_I_and_not_SI>

            </ref_pic_list_reordering>
            <slice_qp_delta>0</slice_qp_delta>
            <if_deblocking_filter_control_present_flag_not_0>
              <disable_deblocking_filter_idc>
                1
              </disable_deblocking_filter_idc>
            </if_deblocking_filter_control_present_flag_not_0>
          </slice_header>
          <skipped_slice_data>
                <!-- This slice contains 233 skipped
                     macroblocks. -->
            <mb_skip_run>233</mb_skip_run>
            <rbsp_trailing_bits>
              <rbsp_stop_one_bit>1</rbsp_stop_one_bit>
              <rbsp_alignment_zero_bit>
                0
              </rbsp_alignment_zero_bit>
            </rbsp_trailing_bits>
          </skipped_slice_data>
        </skipped_slice_layer_without_partitioning_rbsp>
      </coded_slice_of_a_skipped_non_IDR_picture>
    </raw_byte_sequence_payload>
  </nal_unit>
</byte_stream_nal_unit>
```

The following six guidelines are taken into account during the mapping of the syntax of a B slice to the syntax of a skipped P slice. An example stylesheet is shown in Listing D.4 in Appendix D. The process that is responsible for the transformation of the description of a B slice into the description of a skipped B slice omits the second and the fourth step.

1. The `nal_ref_idc` syntax element is not changed, implying that the property whether a slice is used as a reference or not is maintained. This is a first step in keeping the management of reference pictures consistent in case such pictures are involved in the conversion process.
2. The value of the `slice_type` syntax element is changed in order to signal a P slice (emphasized by a bold font in Listing 6.6).
3. The `frame_num` syntax element is merely copied to the target slice header (no reference pictures are removed), as well as all information

related to the display order management (slices are only replaced by skipped ones), all information pertaining to the deblocking filter, and all information concerning memory management control operations.

4. The parameters related to B slices, used in the management of the reference lists and the reconstruction of a slice (weighted prediction), are dropped (denoted by white space in Listing 6.6).

5. To save a number of extra bits, the value of `slice_qp_delta` is initialized to zero as this value is not used during the reconstruction of a skipped slice (indicated by a bold font in Listing 6.6).

6. Finally, `slice_data()` is replaced by a new syntax structure that signals to a decoder that all macroblocks of the corresponding slice are coded as skipped (stressed by a bold font in Listing 6.6).

A BS Schema fragment, describing the syntax of a skipped slice in H.264/AVC, is provided in Listing 6.7. This BS Schema excerpt is required in order to achieve a correct binarization of the slice data. Note that the BSD-driven exploitation of temporal scalability, using placeholder pictures, is only allowed when CAVLC entropy coding is in use. For CABAC requires the use of a context-adaptive coded syntax element to communicate a skipped macroblock run, something that cannot be expressed in the current version of MPEG-21 BSDL (when CAVLC entropy coding is in use, the `mb_skip_run` syntax element, indicating the number of skipped macroblocks, is represented using Unsigned Exponential Golomb coding). This issue could be solved by temporarily changing the entropy coding scheme used[5].

Finally, due to the required complexity for signaling a skipped picture using the H.264/AVC syntax, a rather complex analysis step is needed by a multiplexer to detect the presence of this type of pictures in an elementary H.264/AVC bitstream. On top of that, when CABAC is in use, the multiplexer also needs to be aware of this entropy coding scheme in order to perform a correct bitstream analysis. This is in contrast to MPEG-4 Visual and VC-1, where the detection of placeholder pictures only requires checking the value of the `vop_coded` and `ptype` syntax elements, respectively.

Listing 6.7: BS Schema fragment for skipped slices in H.264/AVC.

```
<xsd:element name="raw_byte_sequence_payload">
  <xsd:complexType>
```

[5]In an H.264/AVC bitstream, the entropy coding scheme can be changed by sending a new PPS, containing an appropriate value for the `entropy_coding_mode_flag` syntax element.

```
    <xsd:choice>
      <xsd:element name="unspecified" type="jvt:PayloadType"
                   minOccurs="0"
                   bs2:if="../jvt:nal_unit_type = 0"/>
      <xsd:element name="coded_slice_of_a_non_IDR_picture"
                   minOccurs="0"
                   bs2:if="../jvt:nal_unit_type = 1">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element
              ref="jvt:slice_layer_without_partitioning_rbsp"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <!-- The following entry is only taken during the
           regeneration of a bitstream using BSDtoBin. -->
      <xsd:element
           name="coded_slice_of_a_skipped_non_IDR_picture"
           minOccurs="0"
           bs2:if="../jvt:nal_unit_type = 1">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element
      ref="jvt:skipped_slice_layer_without_partitioning_rbsp"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <!-- Other NAL unit types. -->
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
<xsd:element
     name="skipped_slice_layer_without_partitioning_rbsp">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="jvt:slice_header"/>
      <xsd:element ref="jvt:skipped_slice_data"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="skipped_slice_data">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element
           name="mb_skip_run" type="jvt:UnsignedExpGolomb"/>
      <xsd:element ref="jvt:rbsp_trailing_bits"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Table 6.1: Properties of different versions of *The New World* movie trailer.

ID	coding pattern	picture rate (Hz)	resolution	#slices/ picture	#NALUs	duration (s)	NALU rate (NALUs/s)	size$_o$[a] (MB)
TNW$_{v,1}$	varying	23.98	848x352	5	17819	148	120.4	37.4
TNW$_{v,2}$	varying	23.98	1280x544	7	24945	148	168.5	110.0
TNW$_{v,3}$	varying	23.98	1904x800	9	32071	148	216.7	163.0
TNW$_{f,1}$	IbBbBbBbP...	23.98	848x352	5	17808	148	120.3	42.9
TNW$_{f,2}$	IbBbBbBbP...	23.98	1280x544	5	17808	148	120.3	78.7
TNW$_{f,3}$	IbBbBbBbP...	23.98	1904x800	5	17808	148	120.3	126.0

[a]Label size$_o$ stands for original file size. TNW$_{v,1}$ (TNW$_{f,1}$) denotes an instance of the movie trailer using a varying (fixed) coding pattern.

6.3 Performance of temporal adaptation in H.264/AVC

In this section, we discuss a number of experiments that target the adaptation of elementary H.264/AVC bitstreams in the temporal domain. The purpose of our experiments is twofold.

1. A first goal is to test the expressive power and performance of an entire BSD-driven content adaptation chain for a set of real-world H.264/AVC bitstreams. In the experiments that were outlined in the previous chapter, bitstreams were used with a limited duration of ten seconds, due to performance issues with the BintoBSD Parser that is available in the MPEG-21 reference software package. However, in this series of experiments, optimized BSD producers are used, which allow the processing of bitstreams with an arbitrary duration.

2. A second goal of this series of experiments is to evaluate the performance of the exploitation of temporal scalability in H.264/AVC bitstreams, using the enhanced content adaptation chain that we have proposed in Section 6.2.2. For example, the BSD transformation step, used to translate the descriptions of certain pictures into descriptions of placeholder pictures, can be considered more complex than a BSD transformation step that only involves the removal of the descriptions of particular pictures.

6.3.1 Methodology

Our experiments aim at the temporal adaptation of two different classes of H.264/AVC bitstreams, both having a fixed picture rate of 23.98 Hz: on the one hand bitstreams with a varying coding pattern, and on the other hand bitstreams with a regular coding structure. The test bitstreams used are all different instances of a single movie trailer, called *The New World*. The most important properties of these bitstreams are provided in Table 6.1.

- The first class of bitstreams was downloaded from a well-known media content site[6], where they are deployed in a real-world simulstore scenario. These bitstreams are characterized by a varying coding pattern. More precisely, a P slice coded picture is *mostly* alternated with a non-reference B slice coded picture. Furthermore, every bitstream contains one SPS and one PPS, as well as one buffering period SEI message. These syntax structures are stored at the start of the test bitstreams.
- The second class of bitstreams was encoded with the H.264/AVC reference software (JM 10.2), using a hierarchical coding pattern. This coding structure, which is visualized in Figure 3.9, contains four temporal levels; it is based on the use of hierarchical B pictures. A QP of 20 was used for the coding of the slices in the base layer; for every temporal enhancement layer, the value of the QP was increased with 2.

The performance analysis was done by breaking up the BSD-based content adaptation chain in its three fundamental building blocks: BSD generation, BSD transformation, and adapted bitstream construction. The focus is put on the real-time adaptation of H.264/AVC bitstreams in the temporal domain. Real-time means that every building block is able to achieve a throughput that is at least as fast as the playback speed of the original media resource. These building blocks may for instance run in a pipelined fashion on different processing nodes in a networked multimedia environment.

For the BSD generation step in this series of experiments, a BFlavor-driven parser is employed for the automatic creation of BSDs, as well as an optimized BintoBSD Parser. This optimized BintoBSD Parser uses the context management attributes that are briefly explained in Appendix C. These attributes, which are defined on top of BSDL-2, provide BintoBSD with the means to create a BSD with a constant and minimal memory consumption on the one hand, and with a constant generation speed on the other hand.

Further, Streaming Transformations for XML (STX; [117]) are introduced as a new BSD transformation technology. STX is intended to be a high-speed, low memory consumption alternative to XSLT as it does not require the construction of a complete in-memory tree[7]. As such, STX is suitable for the

[6]Available online: `http://www.apple.com/trailers/`.

[7]The use of STX is similar to the use of the context management attributes in BSDL: the author is responsible for keeping track what information needs to buffered (STX) or stored in the context (BSDL) for further processing.

Table 6.2: BSD generation using an optimized BintoBSD Parser and BFlavor.

ID	BintoBSD$_m$ Parser[a]				BFlavor			
	throughput (NALUs/s)	MC (MB)	BSD (MB)	BSD$_c$ (KB)	throughput (NALUs/s)	MC (MB)	BSD (MB)	BSD$_c$ (KB)
TNW$_{v,1}$	136.0	1.7	35.0	352	1362.5	0.7	24.1	314
TNW$_{v,2}$	122.3	1.7	45.9	498	829.1	0.7	31.6	352
TNW$_{v,3}$	115.4	1.7	59.8	607	736.0	0.7	41.1	536
TNW$_{f,1}$	124.5	1.7	44.3	326	1164.3	0.7	28.9	308
TNW$_{f,2}$	109.8	1.7	44.2	335	777.5	0.7	29.0	317
TNW$_{f,3}$	97.1	1.7	44.3	332	532.7	0.7	29.0	314

[a]BintoBSD$_m$ denotes the optimized BintoBSD Parser, MC stands for peak heap Memory Consumption, and BSD$_c$ for compressed BSD size.

transformation of large XML documents with a repetitive structure. As shown in Listing 4.10 in Chapter 4, these characteristics are typical for BSDs that describe the high-level structure of elementary video bitstreams. Indeed, several publications have shown that XSLT, as well as a hybrid combination of STX/XSLT, are both unusable in the context of XML-driven video adaptation, due to a respective high memory consumption (see Section 4.5 in Chapter 4) and a high overhead in terms of execution times [27, 38][8].

The performance data were obtained on a PC with an Intel Pentium IV 2.61 GHz CPU and 512 MB of memory. The time measurements were done eleven times, after which an average was taken of the last ten runs in order to take into account the start-up period of the different programs involved. BSDs were compressed using WinRAR 3.0's default text compression algorithm. The anatomy of the H.264/AVC bitstreams was described up to and including the syntax elements of the slice headers, once in MPEG-21 BSDL and once in BFlavor. The STX engine used was the Joost STX processor (version 2005-05-21)[9].

6.3.2 BSD generation

Table 6.2 summarizes the results that were obtained during the generation of BSDs for the two classes of test bitstreams. The BFlavor-based parser outperforms the optimized BintoBSD Parser on the three metrics applied.

[8]STX can also be used in streaming scenarios, which is in contrast to XSLT.

[9]Available online: `http://stx.sourceforge.net/`.

- **Execution times.** The BFlavor-driven parser is faster than real time for all bitstreams used. Indeed, for both classes of bitstreams, the minimum throughput of the parser (532.7 NALUs/s) is higher than the maximum required playback speed (216.7 NALUs/s). Such real-time behavior cannot be observed for the optimized BintoBSD Parser. TNW$_{v,1}$ and TNW$_{f,1}$ are the only bitstreams that can be processed in real time by BintoBSD$_m$.

- **Memory consumption.** The BFlavor-driven parser, as well as the optimized BintoBSD Parser, are characterized by a low memory footprint. Both parsers need less than 2 MB of system memory to generate a BSD for all the bitstreams in the test set.

- **Textual BSD sizes.** The BFlavor-driven parser produces textual BSDs that are smaller than those created by the optimized BintoBSD Parser. This is due to the design of our manually created BS Schema (usable by a BintoBSD and BSDtoBin Parser), which is less optimized than BFlavor's automatically generated BS Schema (only to be used by a BSDto-Bin Parser) for the purpose of readability.

Further, from Table 6.2, we can see a significant difference in throughput of the BFlavor-driven parser and the optimized BintoBSD Parser. This observation can be explained by the fact that BFlavor is at large I/O bound, while the parsing speed of BintoBSD$_m$ is determined by its internal use of XPath and by the amount of I/O operations needed for reading a coded bitstream and for writing a BSD. For example, from a profiling of BintoBSD$_m$ for the parsing of TNW$_{f,1}$, we learned that about 40% of its execution time is spent to I/O operations, while about 10% of its execution time is dedicated to the evaluation of XPath expressions. In this context, it is important to note that the implementation of the I/O model of XFlavor/BFlavor (based on the exchange and processing of integers) can be considered much more efficient than the I/O model of BintoBSD$_m$ (based on the exchange and processing of byte arrays, of which the use may require a lot of conversion steps).

Finally, from Table 6.2, we can also observe that the throughput of the BFlavor-driven parser and the optimized BintoBSD Parser decreases when the file size of the bitstream parsed increases. The numbers in this table make clear that the decrease in throughput is more significant for the BFlavor-based parser than for the optimized BintoBSD Parser. For example, when comparing the parsing speeds achieved for TNW$_{f,1}$ and TNW$_{f,3}$, the throughput decreases with 54% for BFlavor and with 22% for BintoBSD$_m$, while the file size of TNW$_{f,3}$ is three times bigger than the file size of TNW$_{f,1}$. It is likely that

Table 6.3: BSD transformation using STX and adapted bitstream construction using BSDL's format-neutral BSDtoBin Parser.

ID	operation	BSD transformation				bitstream construction		
		throughput (NALUs/s)	MC (MB)	BSD (MB)	BSD_c (KB)	throughput (NALUs/s)	MC (MB)	size_a (MB)
$\text{TNW}_{v,1}$	remove B	1009.5	1.2	14.5	168	606.1	1.9	26.5
$\text{TNW}_{v,2}$	remove B	1092.1	1.2	19.1	232	552.7	1.9	73.7
$\text{TNW}_{v,3}$	remove B	1085.7	1.2	25.0	293	399.2	1.9	109.0
$\text{TNW}_{v,1}$	remove P + B	1354.7	1.2	2.1	31	401.1	2.3	5.7
$\text{TNW}_{v,2}$	remove P + B	1408.2	1.2	3.8	57	290.8	3.7	19.7
$\text{TNW}_{v,3}$	remove P + B	1399.6	1.2	5.6	78	150.6	4.6	32.6
$\text{TNW}_{v,1}$	replace B[a]	619.5	2.7	27.8	226	606.1	1.9	26.6
$\text{TNW}_{v,2}$	replace B	661.7	2.8	36.4	309	559.3	1.9	73.9
$\text{TNW}_{v,3}$	replace B	648.3	2.6	47.5	385	554.9	1.9	109.0
$\text{TNW}_{v,1}$	replace P + B[b]	513.4	1.3	28.9	118	592.0	2.6	5.9
$\text{TNW}_{v,2}$	replace P + B	550.1	1.3	37.8	190	620.5	4.0	19.9
$\text{TNW}_{v,3}$	replace P + B	548.0	1.3	49.1	230	522.3	4.5	33.0
$\text{TNW}_{f,3}$	remove EL[c] 3	835.3	1.2	21.5	159	406.2	2.0	98.4
$\text{TNW}_{f,3}$	remove EL 2 + 3	980.0	1.2	11.0	81	361.2	2.3	65.2
$\text{TNW}_{f,3}$	remove EL 1 + 2 + 3	1098.0	1.3	5.8	41	264.3	2.2	38.6
$\text{TNW}_{f,3}$	replace EL 3	536.5	1.7	36.9	194	515.2	2.1	98.5
$\text{TNW}_{f,3}$	replace EL 2 + 3	444.5	1.3	33.5	121	554.7	2.3	65.3
$\text{TNW}_{f,3}$	replace EL 1 + 2 + 3	447.2	1.3	33.1	84	537.2	2.2	38.8

[a]Non-reference B slices are replaced by non-reference skipped P slices.

[b]Non-reference P and B slices are replaced by non-reference skipped P slices.

[c]EL stands for enhancement layer, BSD_c for compressed BSD size, and size_a for adapted bitstream size.

the XPath dependency of the optimized BintoBSD Parser explains the smaller influence of the file size on the time needed to process a particular bitstream by BintoBSD_m.

To summarize, for this series of tests, the BFlavor-based parser is able to generate a BSD in real time. Its performance is determined by the file size of the input bitstream (i.e., its bit rate) and by the file size of the BSD that is to be written to the hard disk. On the other hand, our current implementation of BSDL's BintoBSD process can only be used for offline BSD generation; it cannot be used for real-time BSD generation. The performance of this parser is determined by the file size of the input bitstream, the amount of XPath expressions in the BS Schema used, and the file size of the BSD that is to be written to the hard disk. However, it is expected that the current implementation of BintoBSD can be significantly improved in terms of its I/O handling.

6.3.3 BSD transformation and adapted bitstream construction

In the context of this research, a number of stylesheets were implemented using STX. These stylesheets are responsible for the transformation of the BSDs, generated by the optimized BintoBSD Parser. The adapted bitstreams were constructed by relying on an optimized version of BSDL's BSDtoBin Parser, making use of buffered I/O instead of byte-based output streams.

In what follows, the semantics and performance of three different transformation steps are outlined in more detail: the exploitation of temporal scalability by dropping slices; the enhanced exploitation of temporal scalability using skipped slices; and the creation of video skims (i.e., video summaries). The latter application relies on the use of placeholder pictures in order to be able to synchronize a number of representative key pictures with an audio stream.

Exploiting temporal scalability by dropping slices

For the first set of test bitstreams (i.e., bitstreams with a varying coding pattern), two STX stylesheets were developed to drop P and/or B slice coded pictures. The adaptation operations are respectively denoted 'remove B' and 'remove P + B' in Table 6.3. The disposal of the pictures in question was guided by relying on the values of the `nal_ref_idc` and/or `slice_type` syntax elements. However, the adapted bitstreams, produced by BSDtoBin after the BSD transformation step, can no longer be processed by a multiplexer in a correct way. This is due to an incorrect timing of the remaining pictures. Note that a compliant H.264/AVC decoder can still decode the adapted bitstreams, as such a decoder is unaware of information in the systems layer.

For the second class of bitstreams (i.e., bitstreams with a fixed and hierarchical coding pattern), three STX stylesheets were written for the removal of the different temporal enhancement layers. The corresponding adaptation operations are denoted 'remove EL 3', 'remove EL 2 + 3', and 'remove EL 1 + 2 + 3' in Table 6.3. As layer numbers are not explicitly embedded in the elementary bitstream syntax, the decision-making process for the disposal of certain pictures was implemented by checking the values of the following syntax elements: `nal_ref_idc`, `slice_type`, and/or `frame_num`. The value of the `gaps_in_frame_num_value_allowed_flag` syntax element in the SPS syntax structure was modified to one, signaling to a decoder that reference pictures were dropped intentionally (such pictures are used in the first and the second enhancement layer). An example STX stylesheet, used for the removal of the second and third temporal enhancement layer, is provided in Listing D.3 in Appendix D. Thanks to the use of a dyadic coding structure, the adapted bitstreams can be further processed by a multiplexer in a correct way.

As shown in Table 6.3, the implementation of the different removal operations can be done efficiently in terms of processing time and memory consumption needed, first at the level of a BSD by making use of STX, and second at the level of the coded H.264/AVC bitstream by making use of BSDL's BSDtoBin Parser. In particular, the BSD transformation step, implemented using STX, is characterized by a constant and low memory usage, which is in contrast to the memory-consuming nature of XSLT.

Further, one can also observe that the BSDtoBin Parser is characterized by a decreasing throughput for the construction of the adapted bitstreams. For example, for bitstream $\text{TNW}_{f,3}$, the throughput decreases from 406 NALUs/s for the construction of a bitstream without the third enhancement layer, to 264 NALUs/s for a bitstream that only contains the base layer. This is due to the fact that the base layer contains more coded bitstream data than the temporal enhancement layers: the base layer is composed of I and P slice coded pictures, while the enhancement layers consist of B slice coded pictures (I and P slices usually contain more coded data than B slices, which implies that their processing requires more I/O operations).

Exploiting temporal scalability using skipped slices

A number of STX stylesheets were developed for the enhanced exploitation of temporal scalability in the two different sets of elementary H.264/AVC bitstreams.

For the first class of bitstreams, skipped P slices were used as a substitute for B slices on the one hand, and as a replacement for P and B slices on the other hand. These adaptation operations are respectively denoted 'replace B' and 'replace P + B' in Table 6.3. A STX stylesheet for replacing B slices by skipped P slices is provided in Listing D.4 in Appendix D. The resulting bitstreams can be further processed by a multiplexer as the total number of pictures remains unchanged.

For the second class of bitstreams, the removal of the pictures in the third enhancement layer was implemented by using skipped B slices as a substitute for the non-reference B slices. The use of skipped P slice coded pictures, which instruct a decoder to output the decoded picture at position 0 in list 0 in the decoded picture buffer, would lead to a wrong output order in this case, i.e. $\text{I}_0 B_4 \text{B}_3 B_4 \text{B}_2 B_4 \text{B}_4 B_4 \text{P}_1$ (decoding order: $\text{I}_0\text{P}_1\text{B}_2\text{B}_3\text{B}_4\text{B}_4\text{B}_4\text{B}_4\text{B}_4$). This is due to the fact that B_4 is the most recent decoded picture in the decoded picture buffer (see Figure 3.9). Consequently, the use of skipped B slices in the third enhancement layer implies that a non-reference B slice coded picture is replaced by a picture that is the result of an interpolation process between the previous and the next picture in output order. It is clear that the quality of the reconstructed picture (e.g., in terms of ghosting artifacts) depends on the amount of motion in the video sequence and on the distance between the reference pictures.

Skipped P slices were used as a substitute for B slices when two or more enhancement layers are involved in the adaptation process. This approach makes it possible to obtain a correct output order after the adaptation step, e.g. $I_0B_2B_2B_2B_2B_2B_2B_2P_1$ in case two enhancement layers are adjusted (decoding order: $I_0P_1B_2B_2B_2B_2B_2B_2B_2$).

Note that the enhanced exploitation of temporal scalability can be considered rather academic for the second class of bitstreams. Indeed, thanks to the dyadic coding structure of the elementary video bitstreams, the removal of one or more temporal enhancement layers results in adapted bitstreams that are still characterized by a fixed but lower picture rate. Such bitstreams can be merely processed by a multiplexer.

Performance data are provided in Table 6.3. The results show that the different BSD transformation steps can be executed in real time and with a limited memory consumption. A similar observation can be made for the behavior of BSDL's BSDtoBin Parser, which is used for the construction of tailored bitstreams (placeholder pictures are taken into account in the throughput computations). Also, as shown in the column with label size$_a$, the bitstream overhead of the skipped slices is typically less than or equal to 0.4 MB. For example, comparing line 1 and line 7 in Table 6.3, the overhead of the skipped slices is 0.1 MB (26.6 MB - 26.5 MB).

Creation of video skims by key frame selection

In the previous sections, we have studied two different adaptation techniques that operate along the temporal axis of H.264/AVC bitstreams, i.e. the exploitation of temporal scalability by dropping particular slices on the one hand and the enhanced exploitation of temporal scalability by using skipped slices on the other hand. In this section, a third approach is discussed towards the temporal adaptation of H.264/AVC bitstreams. In particular, an outline is provided regarding the BSD-based creation of video skims, which are compact abstractions of long video sequences. The major goal of this experiment was to investigate a number of challenges that may occur during the BSD-driven creation of video skims. For instance, think about the automatic generation of shot detection information, embedding additional metadata in a BSD, synchronization of representative key pictures with a corresponding audio stream, and so on.

Video skims are typically created by filtering out representative pictures from a coded bitstream. The production of such summaries can for example be implemented by selecting key pictures that are located near the beginning of a shot (i.e., by selecting I slice coded pictures). Therefore, bitstreams were created with similar properties to the second class of test bitstreams, but using the IbBbBbBb coding pattern instead of the IbBbBbBbP... coding structure. The former coding pattern offers random access at regular picture intervals since every picture in the base layer consists entirely of I slices.

First, a proof-of-concept STX stylesheet was implemented that marks the key pictures in the BSD that are located near the start of a shot. This stylesheet is guided by the shot detection information produced by the IBM MPEG-7 Annotation Tool[10]. Note that the shot detection information could not be used directly: a picture that sits at the start of a shot does not always coincide with a key picture in the coded bitstream, thus requiring a manual mapping from a picture that denotes the beginning of a shot to a nearby key picture.

As shown in Listing 6.8, the STX stylesheet, which can be found in Listing D.8 in Appendix D, embeds the information about the different shots as a number of additional attributes in a BSD[11]: the `shot` attribute indicates whether or not a NAL unit belongs to an I slice coded picture that is positioned near the start of a shot, while the `pic_cnt` attribute acts as a counter that is incremented for every picture in bitstream order (for debugging purposes).

Listing 6.8: Embedding shot information as additional attributes in a BSD.

```
<bitstream xmlns="h264_avc" xmlns:jvt="h264_avc"
         bitstreamURI="the_new_world_h480p_IbBbBbBb.h264">
  <byte_stream>
    <byte_stream_nal_unit pic_cnt="0" shot="false">
      <!-- Sequence Parameter Set. -->
    </byte_stream_nal_unit>
    <byte_stream_nal_unit pic_cnt="0" shot="false">
      <!-- Picture Parameter Set. -->
    </byte_stream_nal_unit>
    <byte_stream_nal_unit pic_cnt="1" shot="true">
      <!-- First coded slice of I_0 (an IDR picture). -->
    </byte_stream_nal_unit>
    <!-- ... -->
    <byte_stream_nal_unit pic_cnt="2" shot="false">
      <!-- First coded slice of I_1 (a non-IDR picture). -->
    </byte_stream_nal_unit>
    <!-- ... -->
```

[10] Available online: `http://www.alphaworks.ibm.com/tech/videoannex`.

[11] More complex metadata can be added to a BSD using BSDL's `bs1:ignore` language feature.

```
    <byte_stream_nal_unit pic_cnt="3" shot="false">
      <!-- First coded slice of B_2 (a non-IDR picture). -->
    </byte_stream_nal_unit>
    <!-- Remaining byte stream NAL units in decoding order. -->
  </byte_stream>
</bitstream>
```

The resulting BSD is subsequently provided as input to a second STX stylesheet. This stylesheet filters out the representative I slice coded pictures and translates all remaining I and B slices into skipped P slices in order to maintain synchronization with the original audio stream (relying on similar logic as used by the stylesheet shown in Listing D.4 in Appendix D). Note that the employment of two separate stylesheets is in line with the needs of a real-world adaptation scenario. For example, the first stylesheet could be executed on a content server, while the second stylesheet could be executed on a content adaptation node in an active content delivery network.

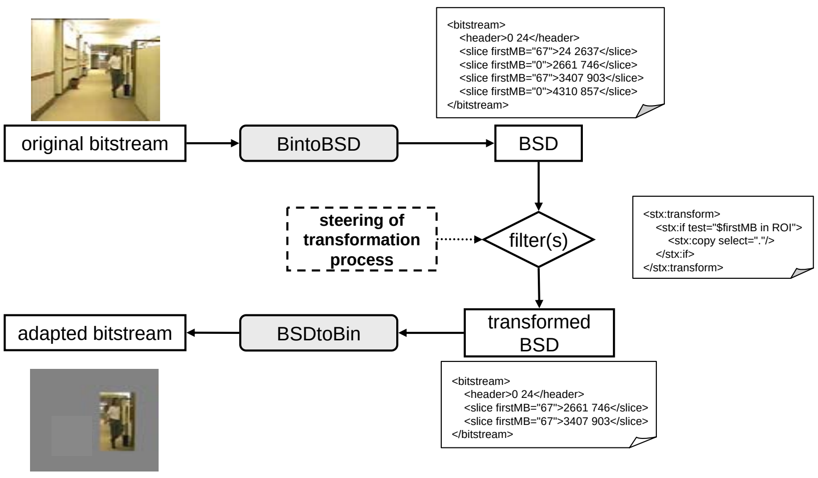
Finally, the creation of a video skim also results in an adapted bitstream with a file size that is significantly smaller than the size of the original bitstream. For example, when $\text{TNW}_{f,1}$ is used with the IbBbBbBb coding pattern, the file size can be reduced from 44.7 MB to 4.40 MB. Consequently, this adaptation technique could be of particular interest for the repurposing of content for constrained usage environments. For instance, an audio stream could be enriched with a limited number of representative pictures that are automatically extracted from a coded video bitstream.

6.4 Enhanced exploitation of ROI scalability

ROI coding can be accomplished in an H.264/AVC bitstream by making use of FMO type 2. As discussed in Chapter 3, this coding feature allows the specification of rectangular areas of interest in a picture, giving an encoder the possibility to make a distinction between a background, i.e. the part of the video pane that does not belong to the ROI, and a foreground, i.e. the ROI. For example, in order to deal with bandwidth fluctuations in an online coding scenario, an encoder may dynamically adjust the bit budget that is assigned to the background, while maintaining the bit rate for the foreground.

In an offline scenario, bit rate savings can be achieved, as well as a trivial form of spatial scalability, by extracting a ROI from an elementary H.264/AVC bitstream. This observation is especially true when the foreground and the background are encoded using the same quality settings (see Table 3.1).

The extraction of a ROI, also known as the exploitation of ROI scalability,



```
<bitstream>
  <header>0 24</header>
  <slice firstMB="67">24 2637</slice>
  <slice firstMB="0">2661 746</slice>
  <slice firstMB="67">3407 903</slice>
  <slice firstMB="0">4310 857</slice>
</bitstream>
```

```
<stx:transform>
  <stx:if test="$firstMB in ROI">
    <stx:copy select="."/>
  </stx:if>
</stx:transform>
```

```
<bitstream>
  <header>0 24</header>
  <slice firstMB="67">2661 746</slice>
  <slice firstMB="67">3407 903</slice>
</bitstream>
```

Figure 6.8: BSD-driven ROI extraction [37].

can be implemented by the disposal of slices that are part of the background. For this purpose, it is necessary to determine whether a particular slice is part of the slice group that constitutes the foreground, or whether this slice belongs to the slice group that constitutes the background. This question can be answered by examining the value of the `first_mb_in_slice` syntax element.

The `first_mb_in_slice` syntax element, which is present in every slice header, conveys the macroblock address of the first macroblock that is part of the slice. Consequently, a slice belongs to the foreground, i.e. the ROI, when the value of the `first_mb_in_slice` parameter falls within the range of macroblock coordinates of the ROI. These coordinates, which may be updated in the course of time, are signaled in the bitstream by a number of syntax elements in the PPS syntax structure. In a BSD-driven content adaptation framework, this decision-making process can be implemented by a stylesheet written in XSLT or STX (see Figure 6.8).

However, a bitstream that is obtained after the exploitation of ROI scalability, is no longer compliant with the first version of the H.264/AVC standard. Indeed, the H.264/AVC specification mandates that all slice groups need to be present in an H.264/AVC bitstream[12]. This implies that an H.264/AVC compliant decoder cannot guarantee the correct decoding of a bitstream in which the background data have been dropped.

Thanks to the independent nature of slices in H.264/AVC, a compliant H.264/AVC decoder only needs a small number of modifications to guaran-

[12]The SVC specification, currently under development, proposes to relax this requirement.

tee the correct decoding of a bitstr(eam in which the background information has been removed. Despite this observation, the creation of invalid bitstreams may be considered a disadvantage of the extraction procedure described above. Therefore, in order to remain compliant with the H.264/AVC specification, it is proposed to eliminate the background in an FMO type 2 encoded bitstream by replacing the background slices with skipped P slices. These skipped P slices instruct a decoder to show the background of the decoded reference picture that is positioned at index 0 in list 0[13]. Consequently, this technique allows the valid exploitation of ROI scalability in H.264/AVC bitstreams.

However, after the extraction of the ROI, the input for the decoder, as well as the corresponding output, is a video sequence at full resolution, containing the background and the foreground. In other words, while the coded data that represent the non-ROI parts of the video sequence were discarded or substituted by placeholder slices, the video pane of the output of the decoder still has the same resolution as the original video sequence.

The aforementioned issue can be solved by relying on the use of frame cropping parameters, which are optionally conveyed by an SPS. By specifying a rectangular region in terms of luma sample coordinates, these syntax elements are able to describe which decoded picture samples are to be outputted by the decoding process. As such, this allows the content adaptation process to map the frame cropping offsets to the area of the ROI [122]. Consequently, it is now possible to create a better match between the output of the decoder and the display capabilities of the target device (e.g., a mobile device with a limited screen resolution). Note that the decoder still has to process the slice group that constitutes the non-ROI. However, as these slices are essentially coded as skipped P slices, the decoding can be considered fairly easy.

Finally, the listing below summarizes the different steps that need to be implemented in order to exploit ROI scalability in a valid way in the first version of the H.264/AVC standard.

- FMO type 2 is used for the coding of one or more ROIs in an H.264/AVC elementary video bitstream.
- BSDL and STX are used for the extraction of one or more multiple ROIs.
- In order to obtain a valid bitstream after ROI extraction, skipped P slices are used as a substitute for the background slices.
- Frame cropping information, conveyed by an SPS, is used for the appropriate rendering of the extracted ROI.

[13]We suppose that the background of the first picture is not dropped by a content adaptation engine.

6.5 Conclusions and original contributions

A placeholder picture can be defined as a picture that is identical to a certain reference picture, or that is constructed by relying on a well-defined interpolation process between different reference pictures. In this chapter, we proposed to make use of placeholder pictures to fill in the gaps that are created in a bitstream due to the removal of certain pictures (temporal scalability). This approach makes it for instance straightforward for a multiplexer to achieve synchronization with other media streams in a particular container format, especially when a varying coding pattern is in use.

The construction of placeholder pictures was outlined for several video coding formats, namely MPEG-1 Video, H.262/MPEG-2 Video, MPEG-4 Visual, VC-1, and H.264/AVC. We subsequently discussed the use of these pictures in a BSD-driven video adaptation framework, providing more details regarding the required modifications. The use of placeholder pictures was stressed for H.264/AVC as the technical design of this specification is the most challenging one for our substitution technique. In this context, it is interesting to note that the BSD-driven approach was used to translate I slices, P slices, and B slices to skipped P slices and skipped B slices, operations that are entirely expressed in the XML domain.

In a next step, we introduced a real-time work flow for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain. The key technologies used were BFlavor for the efficient generation of BSDs, STX for the efficient transformation of BSDs, and BSDL's format-neutral BSDtoBin Parser for the efficient construction of tailored bitstreams. Our approach was validated in several use cases, involving the exploitation of temporal scalability by dropping slices, the exploitation of temporal scalability by relying on placeholder slices, and the creation of video skims. The latter application is made possible by enriching a BSD with additional metadata to steer the BSD transformation process. As an example, an overall pipelined throughput of 264 NALU/s was achieved when exploiting temporal scalability in a High Definition H.264/AVC bitstream by the disposal of all slices in the three temporal enhancement layers, together with a combined memory use of less than 5 MB. The proposed adaptation chain can be considered completely I/O bound.

Finally, we outlined how placeholder slices can be used to fill up the gaps that are created in FMO type 2 encoded H.264/AVC bitstreams due to the removal of certain background slices. Also, a brief discussion was provided regarding the use of the optional frame cropping parameters in an SPS to assist in an appropriate rendering of the extracted ROI. This overall approach makes it possible to obtain valid H.264/AVC bitstreams after the extraction of an ROI, which can be decoded without requiring to modify compliant H.264/AVC decoders.

Our contributions in this research domain can be found in the following publications.

1. Peter Lambert, Dieter Van de Walle, Wesley De Neve, Rik Van de Walle. ROI Scalability in H.264/AVC's Base Specification. Submitted to *Visual Communications and Image Processing 2007 (VCIP 2007)*, San Jose, California, USA, February 2007.

2. Sarah De Bruyne, Wesley De Neve, Koen De Wolf, Davy De Schrijver, Piet Verhoeve, Rik Van de Walle. Temporal Video Segmentation on H.264/AVC Compressed Bitstreams. Accepted for publication in *Proceedings of the International MultiMedia Modeling Conference, MMM 2007*, published by Springer-Verlag in the Lecture Notes in Computer Science series, January 2007.

3. Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Sarah De Bruyne, Rik Van de Walle. Exploitation of Interactive Region of Interest Scalability in Scalable Video Coding by Using an XML-driven Adaptation Framework. In *Proceedings of the 2nd International Conference on Automated Production of Cross Media Content for Multi-channel Distribution*, pages 223–231, Leeds, UK, December 2006.

4. Wesley De Neve, Davy De Schrijver, Davy Van Deursen, Peter Lambert, Rik Van de Walle. Real-Time BSD-driven Adaptation Along the Temporal Axis of H.264/AVC Bitstreams. In *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, Volume 4261, pages 133–143, October 2006.

5. Peter Lambert, Davy De Schrijver, Davy Van Deursen, Wesley De Neve, Yves Dhondt, Rik Van de Walle. A Real-time Content Adaptation Framework for Exploiting ROI Scalability in H.264/AVC. In *Lecture Notes in Computer Science - Advanced Concepts for Intelligent Vision Systems - ACIVS 2006*, Volume 4179, pages 442–453, September 2006.

6. Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Jan De Cock, Rik Van de Walle. On an Evaluation of Transformation Languages in a Fully XML-driven Framework for Video Content Adaptation. In *Proceedings of the 2006 International Conference on Innovative Computing, Information and Control (ICICIC 2006)*, Volume 3, pages 213–216, Beijing, China, August 2006.

7. Peter Lambert, Wesley De Neve, Davy De Schrijver, Yves Dhondt, Rik Van de Walle. Using Placeholder Slices and MPEG-21 BSDL for ROI Extraction in H.264/AVC FMO-encoded Bitstreams. In *Proceedings of International Conference on Signal Processing and Multimedia Applications (SIGMAP 2006)*, pages 9–16, Setúbal, Portugal, August 2006.

8. Sarah De Bruyne, Koen De Wolf, Wesley De Neve, Piet Verhoeve, Rik Van de Walle. Shot Boundary Detection Using Macroblock Prediction Type Information. In *Proceedings of the 7th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 205–208, Incheon, Korea, April 2006.

9. Wesley De Neve, Davy De Schrijver, Dieter Van de Walle, Peter Lambert, Rik Van de Walle. Description-based Substitution Methods for Emulating Temporal Scalability in State-of-the-Art Video Coding Formats. In *Proceedings of the 7th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 83–86, Incheon, Korea, April 2006.

10. Wesley De Neve, Koen De Wolf, Davy De Schrijver, Rik Van de Walle. Using MPEG-4 Scene Description for Offering Customizable and Interactive Multimedia Presentations. In *Proceedings of the 6th Workshop on Image Analysis for Multimedia Interactive Services*, 4 pages on CD-ROM, Montreux, Switzerland, April 2005.

Chapter 7

Conclusions

We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time.

Thomas S. Eliot (1888-1965), Little Gidding (1942).

The last decade has witnessed a significant number of innovative developments in the multimedia ecosystem. Advanced media formats have emerged for the efficient representation, storage, and composition of digital media resources. New network technologies have been devised, wired and wireless, providing access to audio-visual information services such as online music stores, movie download services, and video blogs. A plethora of networked mobile devices has popped up as well, ranging from cell phones to personal entertainment devices, often having sufficient processing power for the playback of multimedia presentations. From these observations, it is clear that the multimedia landscape is characterized by a vast diversity in terms of media formats, network capabilities, and device properties.

The ever-increasing heterogeneity in the multimedia consumption chain poses a number of new challenges. One such challenge is the realization of the *Universal Multimedia Access* paradigm, which is the notion that multimedia content should be accessible at any place, at any time, and with any device. As acknowledged by MPEG, the successful realization of ubiquitous and seamless access to multimedia content requires an appropriate reaction from different knowledge domains.

- The answer of MPEG's coding community consists of the specification of scalable or layered coding schemes. Indeed, the picture rate and spatial resolution of scalable video resources can for instance be adapted in a straightforward way to meet the different constraints that are imposed by a particular usage context (e.g., limitations in terms of available bandwidth and screen resolution).

- The answer of MPEG's metadata community consists of the development of a number of description tools. These tools are for example used to describe the properties of media resources and the capabilities of usage environments. The resulting descriptions enable the construction of a format-agnostic content adaptation system that is able to maximize the user experience, for the consumption of a particular multimedia presentation in a well-defined usage environment.

The research presented in this dissertation is in line with the vision of both communities within MPEG. It essentially investigates and improves the use of a description tool in the context of digital video adaptation.

In Chapter 2, we have outlined the most important concepts and design principles of the state-of-the-art H.264/AVC standard. This specification incorporates the latest advances in standard video coding technology. As shown by our experiments, as well as by other scientific and technical sources, H.264/AVC provides up to 50% bit rate savings for equivalent perceptual quality relative to the performance of prior video coding standards.

The design of the H.264/AVC standard is, besides efficiency, also characterized by a flexibility for use over a broad variety of network types and application domains. In this context, we have studied four content adaptation tools that are part of the initial version of the H.264/AVC specification:

- switching pictures for bitstream switching;
- flexible macroblock ordering for ROI coding;
- data partitioning for realizing coarse grain quality scalability;
- sub-sequences and sub-sequence layers for achieving multi-layered temporal scalability.

These adaptivity provisions, which provide basic means to take into account the constraints of different usage environments, were described in more detail in Chapter 3. The emphasis was put on providing a complete and detailed overview regarding the implementation of multi-layered temporal scalability in H.264/AVC bitstreams. As such, this overview includes an extensive discussion with respect to the use of sub-sequences and sub-sequence layers, coding patterns based on hierarchical B pictures, and SEI messages for communicating the bitstream structure to a bitstream extractor or decoder. Subsequences constrain H.264/AVC's coding flexibility in a minimal way to allow meaningful adaptations in the temporal domain.

More powerful adaptivity features and SEI messages are incorporated in a newly developed amendment to the H.264/AVC specification, which is commonly referred to as H.264/AVC Scalable Video Coding (SVC). This amendment includes explicit support for spatial and quality scalability; temporal adaptivity tools are inherited from the first version of the H.264/AVC standard.

The principles of BSD-driven content adaptation were introduced in Chapter 4. A BSD acts as an intermediate layer between a binary media resource and a content adaptation engine, allowing the development of format-agnostic logic for the adaptation of binary media resources. Two different approaches for BSD-driven content adaptation were studied in more detail: a standardized framework driven by BSDL and a framework based on the use of XFlavor.

Furthermore, the high-level structure of a number of common video coding and container formats was described using MPEG-21 BSDL and XFlavor. The development of a BS Schema for the first version of H.264/AVC, which is at the foundation of a BS Schema for SVC, was discussed in more detail. The construction of this BS Schema exposed a few shortcomings in MPEG-21 BSDL, requiring the use of several non-normative extensions to the BSDL schema language in order to solve the discovered issues.

Besides testing the expressive power of MPEG-21 BSDL and XFlavor, we also evaluated their performance in the context of a number of media formats, targeting applications such as BSD-driven temporal adaptation and demultiplexing. Performance data were particularly provided for a series of experiments targeting the BSD-driven adaptation of VC-1 bitstreams in the temporal domain. Our analysis resulted in the identification of a number of performance bottlenecks that are listed below.

- The slow and memory-consuming generation of BSDs by BSDL's BintoBSD Parser, which is due to the storage of an entire BSD in the system memory to support the evaluation of arbitrary XPath 1.0 expressions. These expressions are embedded in an MPEG-21 BS Schema to steer the parsing behavior of a BintoBSD Parser.

- The verbose BSDs produced by XFlavor-based parsers, as the XFlavor language does not have means for referring to a particular bitstream segment from within a BSD.

- The memory-consuming transformation of BSDs using XSLT.

In Chapter 5, we have proposed BFlavor to solve the aforementioned shortcomings of BSDL and XFlavor. This new description tool is the result of a cross-fertilization between BSDL and XFlavor: it harmonizes both approaches towards XML-driven content adaptation by combining their strengths and by eliminating their weaknesses. In particular, the processing efficiency and expressive power of XFlavor, together with the ability of BSDL to create high-level BSDs, were our key motives for the development of BFlavor. As such, the use of BFlavor-driven BSD producers, which are format-specific but generated automatically by a format-independent process, is an efficient alternative to the use of BSDL's format-neutral but inefficient BintoBSD Parser.

The expressive power and performance of a BFlavor-driven content adaptation chain, compared to tool chains that are either completely based on BSDL or XFlavor, were illustrated by several experiments. One series of experiments targeted the exploitation of multi-layered temporal scalability in H.264/AVC, relying on the combined use of sub-sequences and SEI messages. BFlavor was the only tool to offer an elegant and practical solution for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain.

In Chapter 6, we have outlined the BSD-based construction of placeholder slices and pictures for several video coding formats. These artificial syntax structures allow to eliminate a number of unwanted side-effects that may result from a BSD-driven content adaptation step in the compressed domain. The use of placeholder slices and pictures was discussed in more detail in the context of the BSD-based exploitation of temporal and ROI scalability in the first version of the H.264/AVC standard, respectively providing a solution for synchronization and conformance issues.

Furthermore, Chapter 6 also introduced a real-time work flow for the BSD-driven adaptation of H.264/AVC bitstreams in the temporal domain. The key technologies used were BFlavor for the generation of BSDs, STX for the transformation of BSDs, and BSDL's format-neutral BSDtoBin Parser for the creation of adapted bitstreams. Extensive performance data were provided for several use cases, involving the exploitation of temporal scalability by dropping slices, the exploitation of temporal scalability by relying on placeholder slices, and the creation of video skims. The latter are made possible by enriching a BSD with additional metadata to steer the BSD transformation process.

In this dissertation, we have shown that BSD-driven media resource adaptation can be made possible in an efficient way, using a feasible amount of system memory and processing time. However, an important question remains to be answered: will BSD-driven content adaptation ever be used in real-world multimedia applications for the purpose of format-agnostic content adaptation? A positive answer to this question is dependent on several factors.

1. **Compelling application scenarios:** From the research presented in this dissertation and in the scientific literature, it is clear that the implementation of BSD-driven content adaptation comes with a substantial complexity. This complexity has to be justified by applications that obviously benefit from format-agnostic manipulation of media content. Such scenarios have to involve the use of several scalable media formats to justify the employment of a format-independent content adaptation engine. A possible scenario that can be thought of is the user-centered dissemination of media resources stored in a digital media archive, using different (near-lossless) scalable coding formats for the representation of still images, audio resources, and video resources.

2. **Adoption of scalable coding formats:** A significant number of scalable coding formats exists. However, thus far, scalable coding formats are rarely used in practice for mainstream applications, for reasons identified in the introduction of this dissertation. It is clear that a widespread adoption of scalable coding formats is key to a successful deployment of BSD-driven media resource adaptation.

3. **Textual BSD sizes:** In this research, we have shown that BSD-driven content adaptation can be made possible in an efficient way, in terms of memory consumption and processing time needed. However, the required level of detail for a BSD in order to facilitate certain types of adaptations, and hence its textual size, might pose a problem for particular applications (e.g., in terms of required storage space or bandwidth consumption). Efficient and schema-aware binarization techniques do exist for dealing with verbose BSDs. However, these solutions introduce an additional complexity in the adaptation system.

4. **Semantic metadata:** Another facet that may increase the interest for description-driven content adaptation consists of linking semantic metadata to BSDs. This approach allows to extend the scope of the BSD-driven content adaptation process from the structural level (e.g., exploitation of temporal scalability) to the semantic level (e.g., disposal of violent scenes). Such means are already available in MPEG-21 DIA, but their full potential has not been investigated thoroughly yet.

To conclude this dissertation, we hope that we have convinced the reader that this dissertation, although limited in its scope, contributed to bridging the gap between content and context, supporting the vision that the user, and not the terminal and the network, is to be considered the real point of attention in the multimedia consumption chain.

Appendix A

Syntax and BSD fragments for H.264/AVC

A.1 Introduction

In this appendix, a number of H.264/AVC syntax structures are listed, described in MPEG-21 BSDL, XFlavor, and BFlavor. Due to the length of these syntax descriptions, only a limited number of syntax structures are shown (the BS Schema for H.264/AVC for instance contains about 1200 lines of text). A few BSD fragments are provided as well.

A.2 Syntax fragments

Listing A.1: Excerpt from the manually written BS Schema for H.264/AVC.

```
<xsd:schema
      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:jvt="h264_avc" targetNamespace="h264_avc"
      elementFormDefault="qualified"
      bs2:rootElement="jvt:bitstream">
  <xsd:simpleType name="Stuffing">
    <xsd:restriction base="bs0:fillByte"/>
  </xsd:simpleType>
  <xsd:simpleType name="UnsignedExpGolomb"
       bs0:implementation="datatypes.UnsignedExpGolomb">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
```

```
<xsd:element name="seq_parameter_set_rbsp">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="profile_idc"
                   type="xsd:unsignedByte"/>
      <xsd:element name="constraint_set0_flag" type="b1"/>
      <xsd:element name="constraint_set1_flag" type="b1"/>
      <xsd:element name="constraint_set2_flag" type="b1"/>
      <xsd:element name="reserved_zero_5bits"  type="b5"
                   fixed="0"/>
      <xsd:element name="level_idc" type="xsd:unsignedByte"/>
      <xsd:element name="seq_parameter_set_id"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="log2_max_frame_num_minus4"
                   type="jvt:UnsignedExpGolomb"/>
      <!-- ... -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="pic_parameter_set_rbsp">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="pic_parameter_set_id"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="seq_parameter_set_id"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="entropy_coding_mode_flag"
                   type="b1"/>
      <!-- ... -->
    </xsd:sequence>
  <xsd:complexType>
</xsd:element>
<xsd:element name="sub_seq_layer_characteristics">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="num_sub_seq_layers_minus1"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element
         name="statistics_forloop" maxOccurs="unbounded"
         bs2:nOccurs="./jvt:num_sub_seq_layers_minus1 + 1"/>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element
                 name="accurate_statistics_flag" type="b1"/>
            <xsd:element name="average_bit_rate"
                         type="b16"/>
            <xsd:element name="average_frame_rate"
                         type="b16"/>
          </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="sub_seq_info">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sub_seq_layer_num"
                     type="jvt:UnsignedExpGolomb"/>
        <xsd:element name="sub_seq_id"
                     type="jvt:UnsignedExpGolomb"/>
        <xsd:element name="first_ref_pic_flag" type="b1"/>
        <xsd:element name="leading_non_ref_pic_flag"
                     type="b1"/>
        <xsd:element name="last_pic_flag" type="b1"/>
        <xsd:element name="sub_seq_frame_num_flag"
                     type="b1"/>
        <xsd:element
             name="sub_seq_frame_num_flag" minOccurs="0"
             bs2:if="./jvt:sub_seq_frame_num_flag = 1"
             type="jvt:UnsignedExpGolomb"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="slice_header">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="first_mb_in_slice"
                     type="jvt:UnsignedExpGolomb"/>
        <xsd:element name="slice_type"
                     type="jvt:UnsignedExpGolomb"/>
        <xsd:element name="pic_parameter_set_id"
                     type="jvt:UnsignedExpGolomb"/>
        <xsd:element name="frame_num"
                     type="jvt:FrameNumType"/>
        <xsd:element name="bit_stuffing" minOccurs="0"
                     type="jvt:Stuffing"/>
        <xsd:element name="slice_payload"
                     type="jvt:PayloadType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!-- ... -->
</xsd:schema>
```

Listing A.2: Excerpt from a syntax description in XFlavor for H.264/AVC.

```
class UnsignedExpGolomb {
  // The expressive power of XFlavor allows the processing of
  // syntax elements that are represented with Unsigned and
  // Signed Exponential Golomb coding. This is in contrast to
  // MPEG-21 BSDL: an external procedural object is required -
  // this is, a Java class - to process these syntax elements.
  int leadingzeros = 0;
  while ( nextbits(leadingzeros + 1) == 0 )
    leadingzeros++;
  int length = leadingzeros * 2 + 1;
  bit(length) ue_code;
  int value = ue_code - 1;
  // Use of a verbatim code to write the decoded value to the
  // BSD (for debugging purposes).
  %g.j{XML.WriteXmlElement("<ue_value bitLen="0">" +
                           value + "</ue_value>");%g.j}
}

class Seq_parameter_set_rbsp {
  // profile_idc is a parsable variable: it contains a parse
  // length specification.
  bit(8) profile_idc;
  bit(1) constraint_set0_flag;
  bit(1) constraint_set1_flag;
  bit(1) constraint_set2_flag;
  bit(5) reserved_zero_5bits;
  bit(8) level_idc;
  UnsignedExpGolomb seq_parameter_set_id;
  UnsignedExpGolomb log2_max_frame_num_minus4;
  //...
}

class Pic_parameter_set_rbsp {
  UnsignedExpGolomb pic_parameter_set_id;
  UnsignedExpGolomb seq_parameter_set_id;
  bit(1) entropy_coding_mode_flag;
  //...
}

class Sub_seq_layer_characteristics {
  UnsignedExpGolomb num_sub_seq_layers_minus1;
  // layer is a non-parsable variable: it does not
  // contain a parse length specification.
  int layer;
  for (layer = 0; layer <
       num_sub_seq_layers_minus1.value + 1; layer++) {
    bit(1) accurate_statistics_flag;
    bit(16) average_bit_rate;
    bit(16) average_frame_rate;
  }
}

class Sub_seq_info {
  UnsignedExpGolomb sub_seq_layer_num;
  UnsignedExpGolomb sub_seq_id;
  bit(1) first_ref_pic_flag;
  bit(1) leading_non_ref_pic_flag;
  bit(1) last_pic_flag;
  bit(1) sub_seq_frame_num_flag;
  if ( sub_seq_frame_num_flag )
    UnsignedExpGolomb sub_seq_frame_num;
}

// sps, pps, nal_unit_type, and nal_ref_idc are class
// parameters.
class Slice_header(Seq_parameter_set_rbsp sps,
                   Pic_parameter_set_rbsp pps,
                   int nal_unit_type,
                   int nal_ref_idc) {
  UnsignedExpGolomb first_mb_in_slice;
  UnsignedExpGolomb slice_type;
  UnsignedExpGolomb pic_parameter_set_id;
  // Note the elegant computation of the length of frame_num
  // in terms of a number of bits. This is in contrast to the
  // use of xsd:union/bs2:ifUnion in MPEG-21 BSDL in order to
  // express this calculation.
  bit(sps.log2_max_frame_num_minus4.value + 4) frame_num;
  // numbits() is a built-in operator of XFlavor. It is used to
  // obtain the total number of bits read/written so far
  bit(8 - (numbits() % 8)) stuffbits;
  // XFlavor_Payload is an own-developed class that stores the
  // the remaining bitstream data in the BSD. Every four bytes
  // in the payload are mapped to an unsigned 32-bits integer.
  // Such an approach is necessary to separate the BSD
  // generation and transformation processes.
  XFlavor_Payload payload;
}

// ...
```

Listing A.3: Excerpt from a syntax description in BFlavor for H.264/AVC.

```
// The base class Encoded, providing a mapping to
// bs0:implementation.
// class Encoded {
//  // Data member value contains the decoded value that is to
//  // be placed in the instance document at BSD generation
//  // time.
//      int value;
// }

// Use of the newly introduced verbatim codes.
%targetns{h264_avc%targetns}
%ns{jvt%ns}
%root{bitstream%root}

// The class UnsignedExpGolomb is derived from the base class
// Encoded. Syntax elements represented with Unsigned
// Exponential Golomb coding are parsed by a BFlavor-based
// parser. BSDtoBin relies on an external procedural object to
// binarize such elements as the first version of MPEG-21 BSDL
// does not have built-in support for this datatype.
class UnsignedExpGolomb extends Encoded {
  int length = 0;
  int leadingzeros = 0;
  while ( nextbits(leadingzeros + 1) == 0 )
    leadingzeros++;
  length = leadingzeros * 2 + 1;
  bit(length) ue_code;
  // value is defined in the base class Encoded.
  value = ue_code - 1;
}

class Seq_parameter_set_rbsp {
  bit(8) profile_idc;
  bit(1) constraint_set0_flag;
  bit(1) constraint_set1_flag;
  bit(1) constraint_set2_flag;
  bit(5) reserved_zero_5bits;
  bit(8) level_idc;
  UnsignedExpGolomb seq_parameter_set_id;
  UnsignedExpGolomb log2_max_frame_num_minus4;
  //...
}

class Pic_parameter_set_rbsp {
  UnsignedExpGolomb pic_parameter_set_id;
  UnsignedExpGolomb seq_parameter_set_id;
  bit(1) entropy_coding_mode_flag;
  //...
}

class Sub_seq_info {
  UnsignedExpGolomb sub_seq_layer_num;
  UnsignedExpGolomb sub_seq_id;
  bit(1) first_ref_pic_flag;
  bit(1) leading_non_ref_pic_flag;
  bit(1) last_pic_flag;
  bit(1) sub_seq_frame_num_flag;
  if ( sub_seq_frame_num_flag )
    UnsignedExpGolomb sub_seq_frame_num;
}

class Sub_seq_layer_characteristics {
  UnsignedExpGolomb num_sub_seq_layers_minus1;
  int layer;
  for (layer = 0; layer <
        num_sub_seq_layers_minus1.value + 1; layer++) {
    bit(1) accurate_statistics_flag;
    bit(16) average_bit_rate;
    bit(16) average_frame_rate;
  }
}

class Slice_header(Seq_parameter_set_rbsp sps,
                   Pic_parameter_set_rbsp pps,
                   int nal_unit_type,
                   int nal_ref_idc) {
  UnsignedExpGolomb first_mb_in_slice;
  UnsignedExpGolomb slice_type;
  UnsignedExpGolomb pic_parameter_set_id;
  bit(sps.log2_max_frame_num_minus4.value + 4) frame_num;

  // Use of the newly introduced operator align().
  align();

  // Skip the remaining bitstream data, i.e. create a reference
  // to the bitstream segment skipped. This approach is similar
  // to the one used in MPEG-21 BSDL; it makes it possible to
  // obtain compact BSDs.
  byteRange (3) payload[2] = {0x000000, 0x000001};
}

// ...
```

Listing A.4: Excerpt from a syntax description in MPEG-21 BSDL for H.264/AVC, automatically generated using the BFlavorc translator (i.e., the enhanced version of the Flavorc translator).

```
<xsd:schema
     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:jvt="h264_avc" targetNamespace="h264_avc"
   elementFormDefault="qualified">
<xsd:simpleType name="UnsignedExpGolomb"
                bs0:implementation="UnsignedExpGolomb">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<xsd:complexType name="Seq_parameter_set_rbsp">
  <xsd:sequence>
    <xsd:element name="profile_idc" type="b8"/>
    <xsd:element name="constraint_set0_flag" type="b1"/>
    <xsd:element name="constraint_set1_flag" type="b1"/>
    <xsd:element name="constraint_set2_flag" type="b1"/>
    <xsd:element name="reserved_zero_5bits" type="b5"/>
    <xsd:element name="level_idc" type="b8"/>
    <xsd:element name="seq_parameter_set_id"
                 type="jvt:UnsignedExpGolomb"/>
    <xsd:element name="log2_max_frame_num_minus4"
                 type="jvt:UnsignedExpGolomb"/>
    <!-- ... -->
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Pic_parameter_set_rbsp">
  <xsd:sequence>
    <xsd:element name="pic_parameter_set_id"
                 type="jvt:UnsignedExpGolomb"/>
    <xsd:element name="seq_parameter_set_id"
                 type="jvt:UnsignedExpGolomb"/>
    <xsd:element name="entropy_coding_mode_flag" type="b1"/>
    <!-- ... -->
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Sub_seq_info">
  <xsd:sequence>
    <xsd:element name="sub_seq_layer_num"
                 type="jvt:UnsignedExpGolomb"/>
    <xsd:element name="sub_seq_id"
                 type="jvt:UnsignedExpGolomb"/>
    <xsd:element name="first_ref_pic_flag" type="b1"/>
    <xsd:element name="leading_non_ref_pic_flag" type="b1"/>
    <xsd:element name="last_pic_flag" type="b1"/>
    <xsd:element name="sub_seq_frame_num_flag" type="b1"/>
    <xsd:sequence minOccurs="0">
      <xsd:element name="sub_seq_frame_num"
                   type="jvt:UnsignedExpGolomb"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Sub_seq_layer_characteristics">
  <xsd:sequence>
```

```
      <xsd:element name="num_sub_seq_layers_minus1"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="accurate_statistics_flag"
                     type="b1"/>
        <xsd:element name="average_bit_rate" type="b16"/>
        <xsd:element name="average_frame_rate" type="b16"/>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Slice_header">
    <xsd:sequence>
      <xsd:element name="first_mb_in_slice"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="slice_type"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="pic_parameter_set_id"
                   type="jvt:UnsignedExpGolomb"/>
      <xsd:element name="frame_num" type="jvt:unionType"/>
      <xsd:element name="stuffbits" type="bs0:fillByte"/>
      <xsd:element name="payload" type="jvt:Payload"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- ... -->
</xsd:schema>
```

A.3 BSD fragments

Listing A.5: Excerpt from a BSD for an H.264/AVC bitstream, generated by a BintoBSD Parser. For the syntax structures shown, there are no differences with the BSD that is generated by a BFlavor-driven parser.

```
<bitstream
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="h264_avc h264_avc_bsdl.xsd"
  bs1:bitstreamURI="foreman_SEI.h264"
  xmlns="h264_avc" xmlns:jvt="h264_avc">
  <!-- ... -->
  <byte_stream_nal_unit>
    <zero_byte>0</zero_byte>
    <startcode>000001</startcode>
    <nal_unit>
      <forbidden_zero_bit>0</forbidden_zero_bit>
      <nal_ref_idc>0</nal_ref_idc>
      <nal_unit_type>6</nal_unit_type>
      <raw_byte_sequence_payload>
        <sei_rbsp>
          <sei_message>
            <last_payload_type_byte>10</last_payload_type_byte>
            <last_payload_size_byte>1</last_payload_size_byte>
            <sei_payload>
              <sub_seq_info>
                <sub_seq_layer_num>0</sub_seq_layer_num>
                <sub_seq_id>0</sub_seq_id>
                <first_ref_pic_flag>0</first_ref_pic_flag>
                <leading_non_ref_pic_flag>
                  0
                </leading_non_ref_pic_flag>
                <last_pic_flag>0</last_pic_flag>
                <sub_seq_frame_num_flag>
                  0
                </sub_seq_frame_num_flag>
              </sub_seq_info>
              <stuffbits>2</stuffbits>
            </sei_payload>
          </sei_message>
          <rbsp_stop_one_bit>1</rbsp_stop_one_bit>
          <stuffbits>0</stuffbits>
        </sei_rbsp>
      </raw_byte_sequence_payload>
    </nal_unit>
  </byte_stream_nal_unit>
  <byte_stream_nal_unit>
    <startcode>000001</startcode>
    <nal_unit>
      <forbidden_zero_bit>0</forbidden_zero_bit>
      <nal_ref_idc>3</nal_ref_idc>
      <nal_unit_type>5</nal_unit_type>
      <raw_byte_sequence_payload>
        <slice_layer_without_partitioning_rbsp>
          <slice_header>
            <first_mb_in_slice>0</first_mb_in_slice>
            <slice_type>7</slice_type>
            <pic_parameter_set_id>0</pic_parameter_set_id>
            <frame_num xsi:type="b4">0</frame_num>
            <stuffbits>15</stuffbits>
            <payload>37 72300</payload>
          </slice_header>
        </slice_layer_without_partitioning_rbsp>
      </raw_byte_sequence_payload>
    </nal_unit>
  </byte_stream_nal_unit>
  <!-- ... -->
</bitstream>
```

Listing A.6: Excerpt from a BSD for an H.264/AVC bitstream, generated by an XFlavor-based parser.

```
<bitstream
  xmlns="http://www.ee.columbia.edu/flavor"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- ... -->
  <byte_stream_nal_unit>
    <zero_byte bitLen="8">0</zero_byte>
    <startcode bitLen="24">1</startcode>
    <nal_unit>
      <forbidden_zero_bit bitLen="1">0</forbidden_zero_bit>
      <nal_ref_idc bitLen="2">0</nal_ref_idc>
      <nal_unit_type bitLen="5">6</nal_unit_type>
      <raw_byte_sequence_payload>
        <sei_rbsp>
          <sei_message>
            <last_payload_type_byte bitLen="8">
              10
            </last_payload_type_byte>
            <last_payload_size_byte bitLen="8">
              1
            </last_payload_size_byte>
            <sei_payload>
              <sub_seq_info>
                <sub_seq_layer_num>
                  <ue_code bitLen="1">1</ue_code>
                  <ue_value bitLen="0">0</ue_value>
                </sub_seq_layer_num>
                <sub_seq_id>
                  <ue_code bitLen="1">1</ue_code>
                  <ue_value bitLen="0">0</ue_value>
                </sub_seq_id>
                <first_ref_pic_flag bitLen="1">
                  0
                </first_ref_pic_flag>
                <leading_non_ref_pic_flag bitLen="1">
                  0
                </leading_non_ref_pic_flag>
                <last_pic_flag bitLen="1">0</last_pic_flag>
                <sub_seq_frame_num_flag bitLen="1">
                  0
                </sub_seq_frame_num_flag>
              </sub_seq_info>
              <stuffbits bitLen="3">2</stuffbits>
            </sei_payload>
          </sei_message>
          <rbsp_stop_one_bit bitLen="1">1</rbsp_stop_one_bit>
          <stuffbits bitLen="1">0</stuffbits>
        </sei_rbsp>
      </raw_byte_sequence_payload>
    </nal_unit>
  </byte_stream_nal_unit>
  <byte_stream_nal_unit>
    <startcode bitLen="24">1</startcode>
    <nal_unit>
      <forbidden_zero_bit bitLen="1">0</forbidden_zero_bit>
      <nal_ref_idc bitLen="2">3</nal_ref_idc>
      <nal_unit_type bitLen="5">5</nal_unit_type>
      <raw_byte_sequence_payload>
        <slice_layer_without_partitioning_rbsp>
          <slice_header>
            <first_mb_in_slice>
              <ue_code bitLen="1">1</ue_code>
              <ue_value bitLen="0">0</ue_value>
            </first_mb_in_slice>
            <slice_type>
              <ue_code bitLen="7">8</ue_code>
              <ue_value bitLen="0">7</ue_value>
            </slice_type>
            <pic_parameter_set_id>
              <ue_code bitLen="1">1</ue_code>
              <ue_value bitLen="0">0</ue_value>
            </pic_parameter_set_id>
            <frame_num bitLen="11">0</frame_num>
            <stuffbits bitLen="6">15</stuffbits>
            <payload>
              <payload bitLen="32">453763072</payload>
              <!-- ... -->
            </payload>
          </slice_header>
        </slice_layer_without_partitioning_rbsp>
      </raw_byte_sequence_payload>
    </nal_unit>
  </byte_stream_nal_unit>
  <!-- ... -->
</bitstream>
```

Appendix B

BS Schemata for MPEG media formats

B.1 Introduction

This appendix provides a visualization of a number of BS Schemata[1], describing the most important syntax structures of MPEG-1 Video, MPEG-1 Systems, H.262/MPEG-2 Video, MPEG-2 Systems, MPEG-4 Visual, and H.264/AVC. For each of these formats, a brief explanation is provided regarding the use of the context management attributes. These attributes, which are outlined in Appendix C, allow an efficient generation of BSDs using a format-unaware BintoBSD Parser.

B.2 Features BS Schemata

B.2.1 MPEG-1 Video

The level of detail of the BS Schema for MPEG-1 Video allows discovering the slices of a picture in an elementary MPEG-1 Video bitstream. Sequence headers, group of pictures headers, and pictures are embedded in an abstract parse unit `pu`. This is clarified by Figure B.1 (see Section B.3 in this appendix). The context management is mainly done at the level of a `pu`, which is the fundamental unit of parsing for a BintoBSD Parser in the context of the MPEG-1 Video coding format.

[1]The visualizations were created using XMLSpy (`http://www.altova.com/`).

B.2.2 MPEG-1 Systems

The level of detail of the BS Schema for MPEG-1 Systems allows identifying the packs in a compliant bitstream. This is clarified by Figure B.2. The parsing of the Packetized Elementary Stream packets (PES packets) is implemented using BSDL's `bs2:length` facet. Parsing based on searching for system start codes is not reliable due to their possible emulation in the payload of a PES packet. The context management is mainly done at the level of a pack, the fundamental unit of parsing in the context of MPEG-1 Systems.

B.2.3 H.262/MPEG-2 Video

The level of detail of the BS Schema for H.262/MPEG-2 Video allows discovering the slices of a picture in an elementary H.262/MPEG-2 Video bitstream. Sequence headers, sequence extensions, group of pictures headers, and pictures are embedded in an abstract parse unit `pu`. This is clarified by Figure B.3. The context management is mainly done at the level of this fundamental unit of parsing for a BintoBSD Parser (in the context of the H.262/MPEG-2 Video format).

Note the large similarity between the BS Schema for MPEG-1 Video and the BS Schema for H.262/MPEG-2 Video. This is due to the fact that the syntax of MPEG-1 Video can be considered a subset of the syntax of H.262/MPEG-2 Video. Besides a different slice definition and a number of minor differences at the macroblock level, the H.262/MPEG-2 Video specification also has tools for dealing with interlaced video and scalable video. However, the scalable coding tools of H.262/MPEG-2 Video are not used in practice.

B.2.4 MPEG-2 Systems

The level of detail of the BS Schema for MPEG-2 Systems allows discovering the packs in an MPEG-2 Program Stream on the one hand, and retrieving the transport packets from an MPEG-2 Transport Stream on the other hand. This is clarified by Figure B.4, Figure B.5, Figure B.6, and Figure B.7. Again, note the similarity between an MPEG-1 Systems stream and an MPEG-2 Program Stream. The context management is mainly done at the level of a pack for MPEG-2 Program Streams and at the level of a transport packet for MPEG-2 Transport Streams.

B.2.5 MPEG-4 Visual

The level of detail of the BS Schema for MPEG-4 Visual allows discovering the Video Object Planes (VOPs) in bitstreams compliant with the Advanced Simple Profile (ASP). Similar to MPEG-1 Video and H.262/MPEG-2 Video, the different headers are embedded in an abstract parse unit `pu`. This is clarified by Figure B.8. The context management is mainly done at the level of this fundamental unit of parsing for a BintoBSD Parser.

B.2.6 H.264/AVC

The level of detail of the BS Schema for the first version of H.264/AVC allows parsing all syntax elements up to and including the `slice_header` syntax structure. The most important syntax structures of H.264/AVC are illustrated by Figure B.9, Figure B.10, Figure B.11, Figure B.12, Figure B.13, and Figure B.14. Note that this BS Schema is at the foundation of a BS Schema that describes the high-level syntax of bitstreams compliant with the SVC amendment of H.264/AVC.

Further, although the design of the BS Schema is in line with the generic template as discussed in Chapter 4, the context management in the BS Schema for MPEG-4 AVC can be considered complex due to the pointer-based relationship between a slice header, a PPS, and an SPS on the one hand, and due to the possible occurrence of multiple PPSs and SPSs in an H.264/AVC bitstream on the other hand.

B.3 Visualization of the BS Schemata

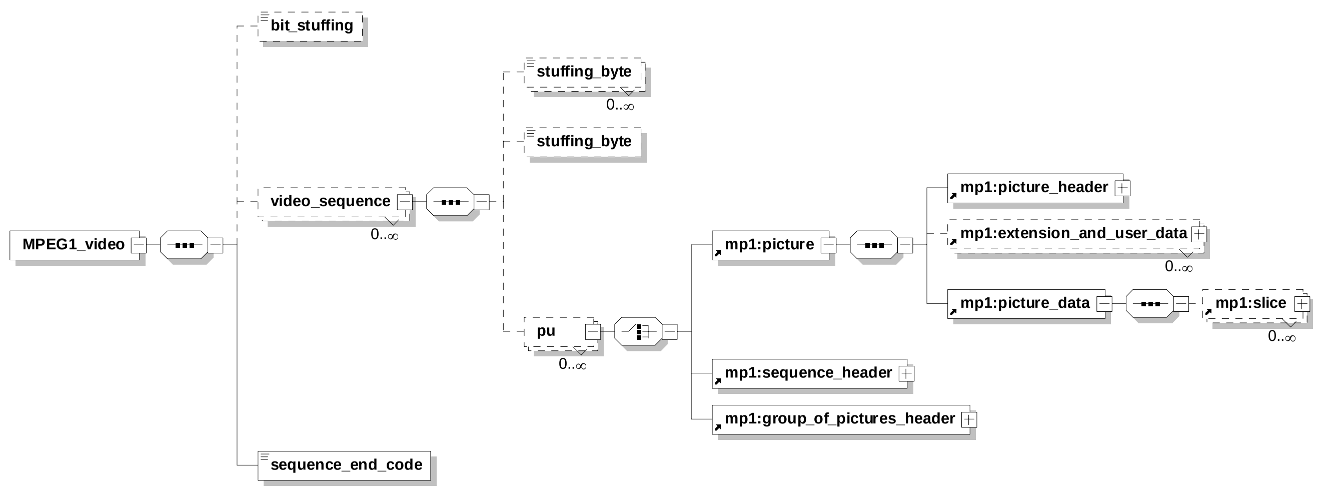


Figure B.1: Excerpt of a BS Schema for the high-level syntax of MPEG-1 Video.

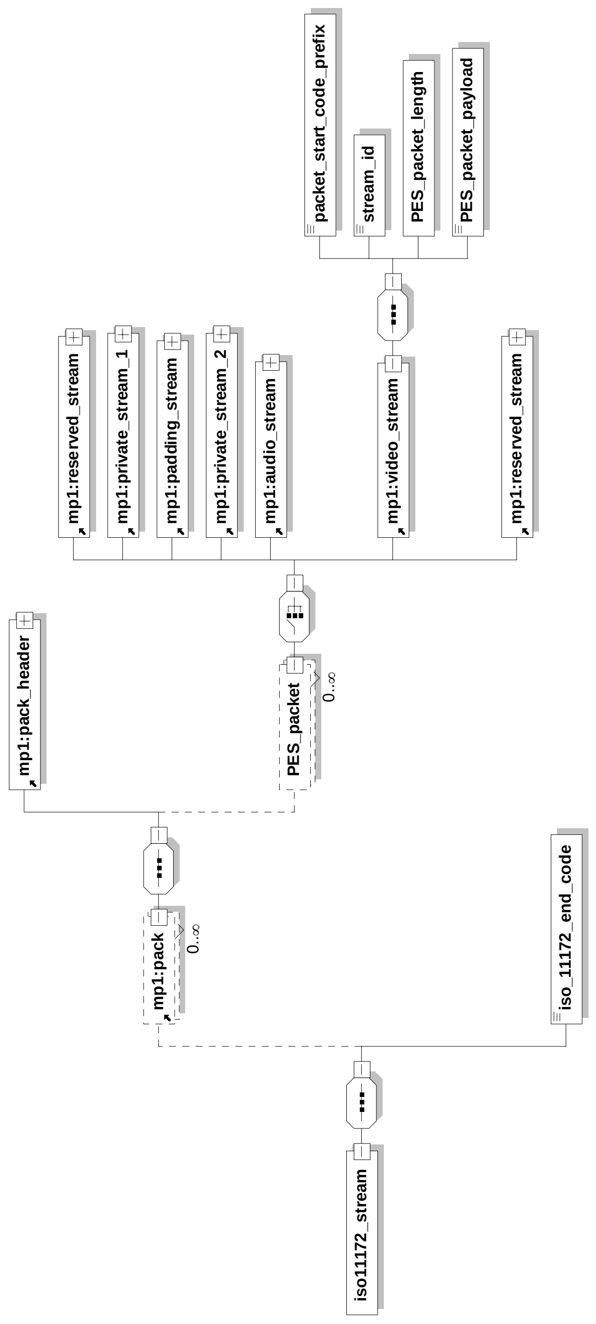


Figure B.2: Excerpt of a BS Schema for the high-level syntax of MPEG-1 Systems.

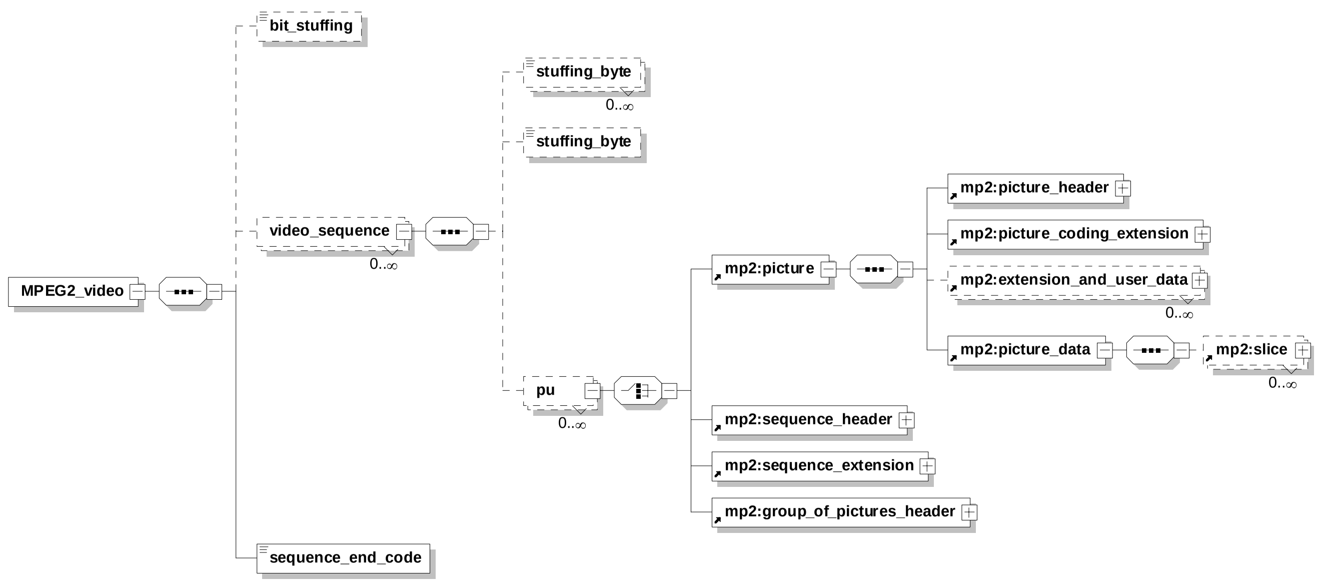


Figure B.3: Excerpt of a BS Schema for the high-level syntax of MPEG-2 Video.

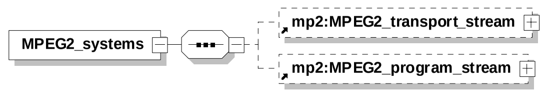


Figure B.4: Overall structure of the BS Schema for MPEG-2 Systems.

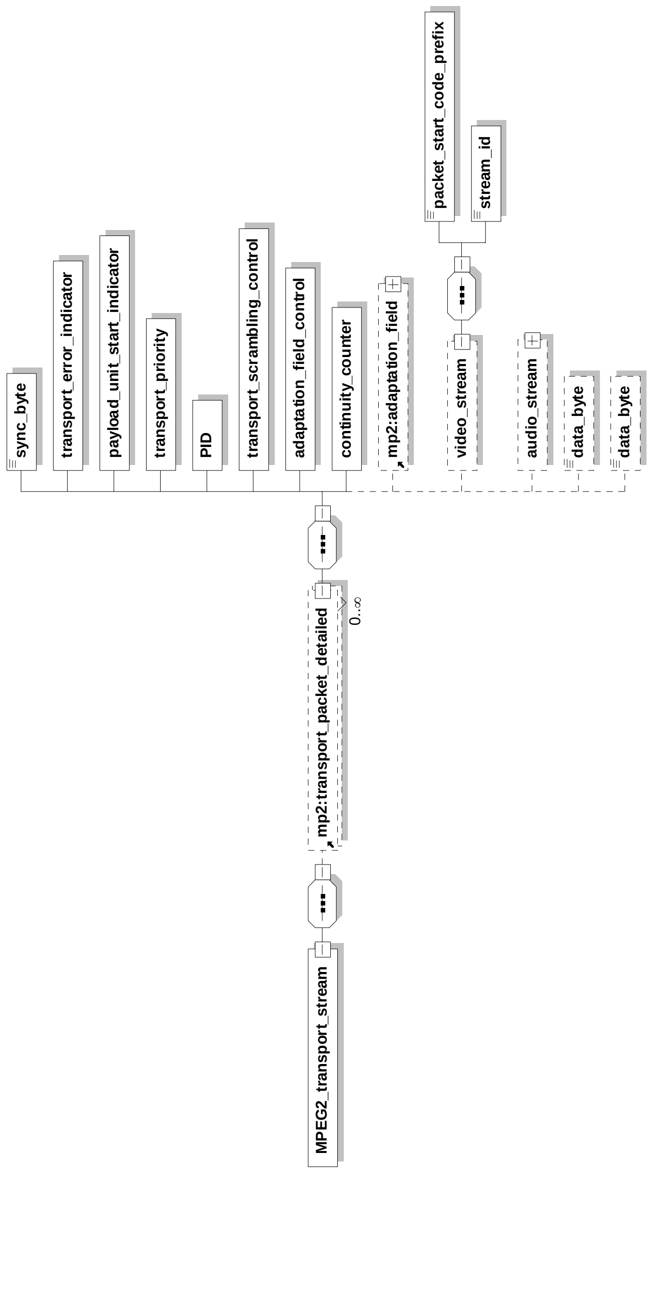


Figure B.5: Excerpt of a BS Schema for the high-level syntax of an MPEG-2 Transport Stream.

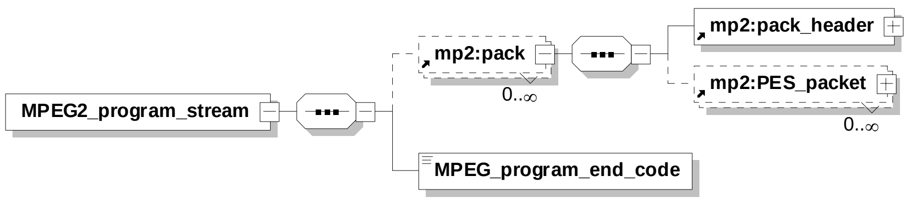


Figure B.6: Excerpt of a BS Schema for the high-level syntax of an MPEG-2 Program Stream.

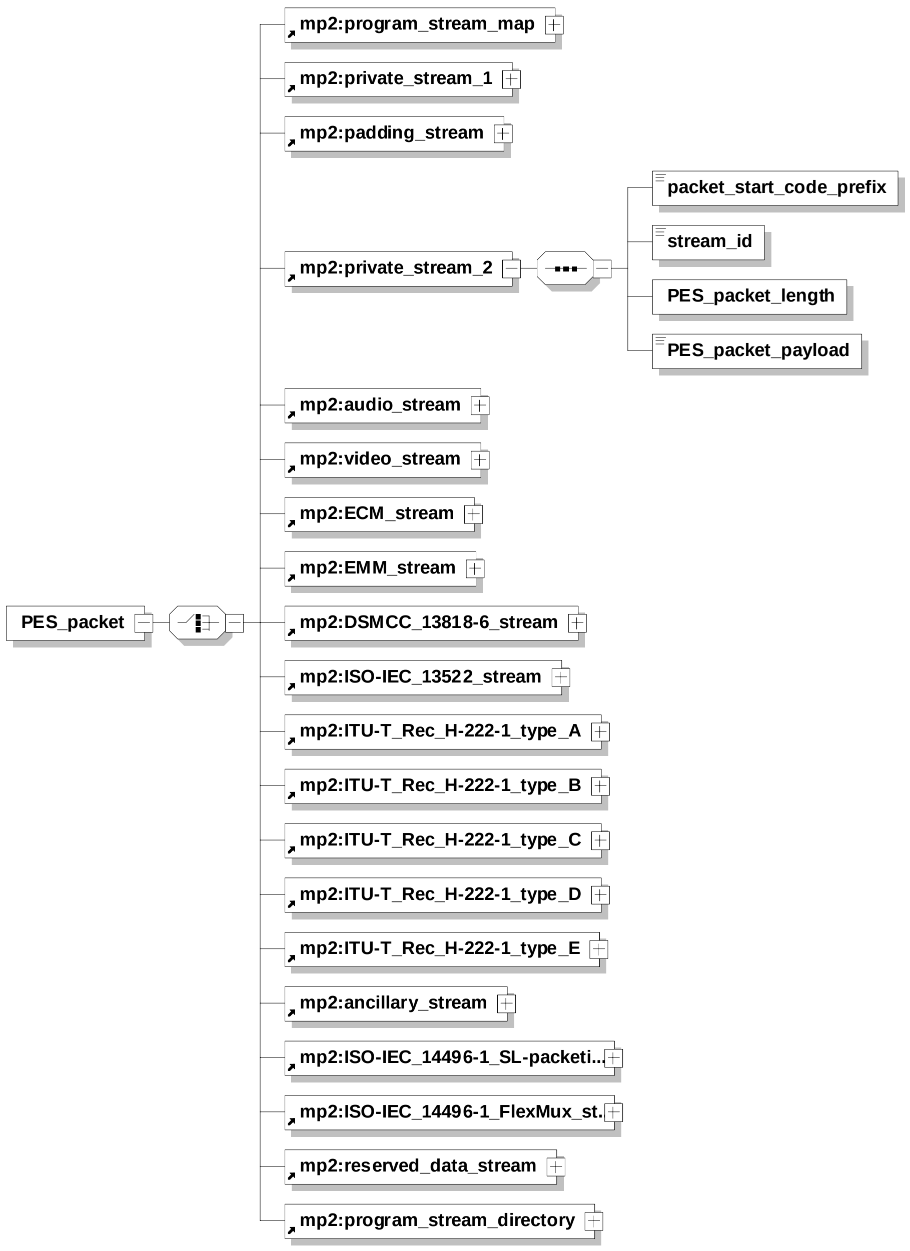


Figure B.7: Excerpt of a BS Schema for the high-level syntax of an MPEG-2 Program Stream, illustrating the structure of a PES packet.

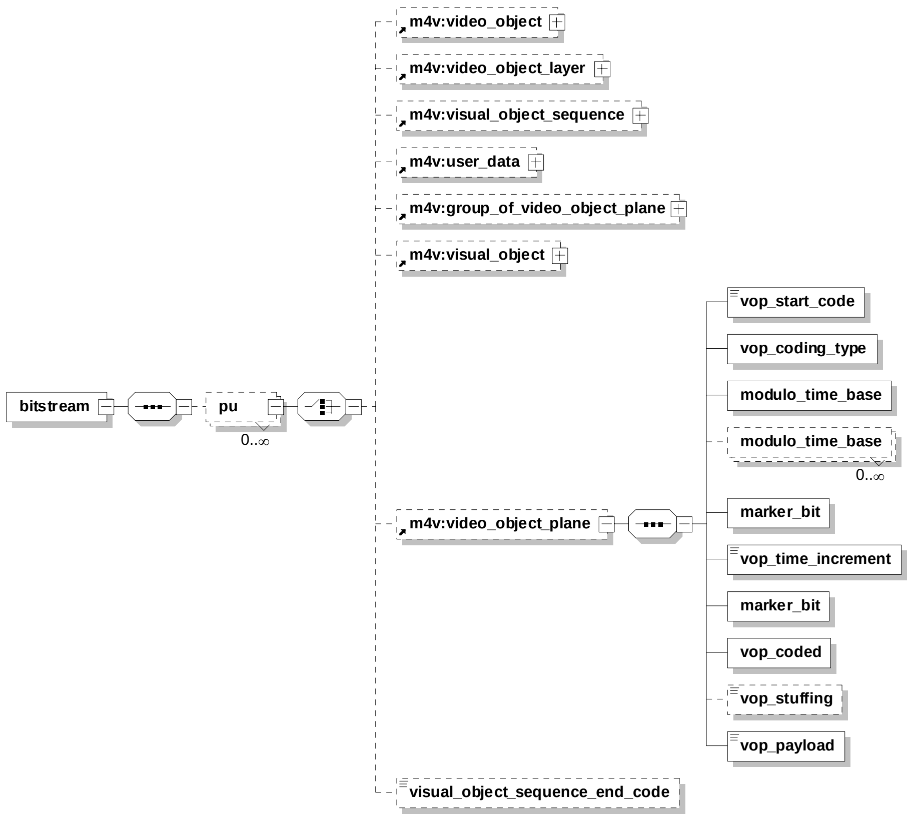


Figure B.8: Excerpt of a BS Schema for the high-level syntax of MPEG-4 Visual.

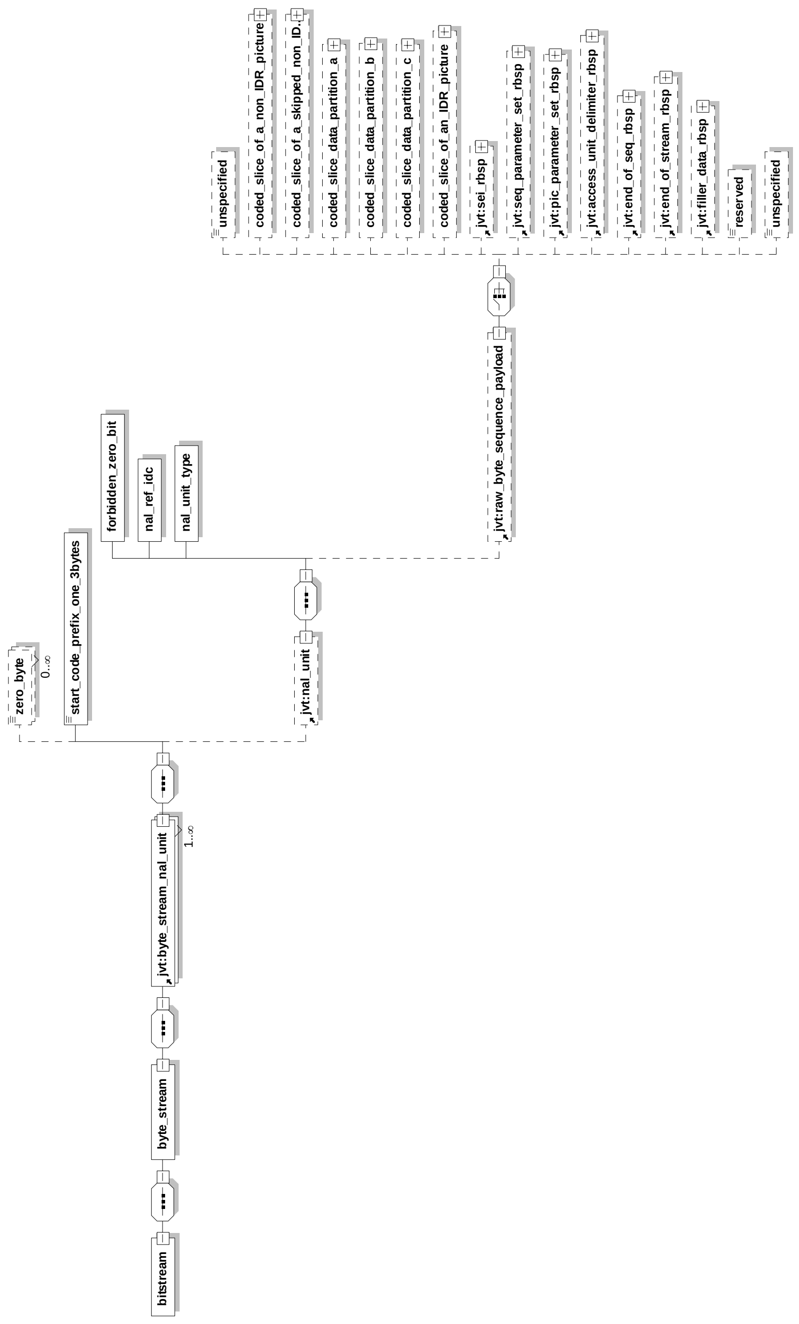

bitstream
byte_stream
jvt:byte_stream_nal_unit
1..∞
zero_byte
0..∞
start_code_prefix_one_3bytes
jvt:nal_unit
forbidden_zero_bit
nal_ref_idc
nal_unit_type
jvt:raw_byte_sequence_payload
unspecified
coded_slice_of_a_non_IDR_picture
coded_slice_of_a_skipped_non_ID...
coded_slice_data_partition_a
coded_slice_data_partition_b
coded_slice_data_partition_c
coded_slice_of_an_IDR_picture
jvt:sei_rbsp
jvt:seq_parameter_set_rbsp
jvt:pic_parameter_set_rbsp
jvt:access_unit_delimiter_rbsp
jvt:end_of_seq_rbsp
jvt:end_of_stream_rbsp
jvt:filler_data_rbsp
reserved
unspecified

Figure B.9: Excerpt of a BS Schema for the high-level syntax of H.264/AVC.

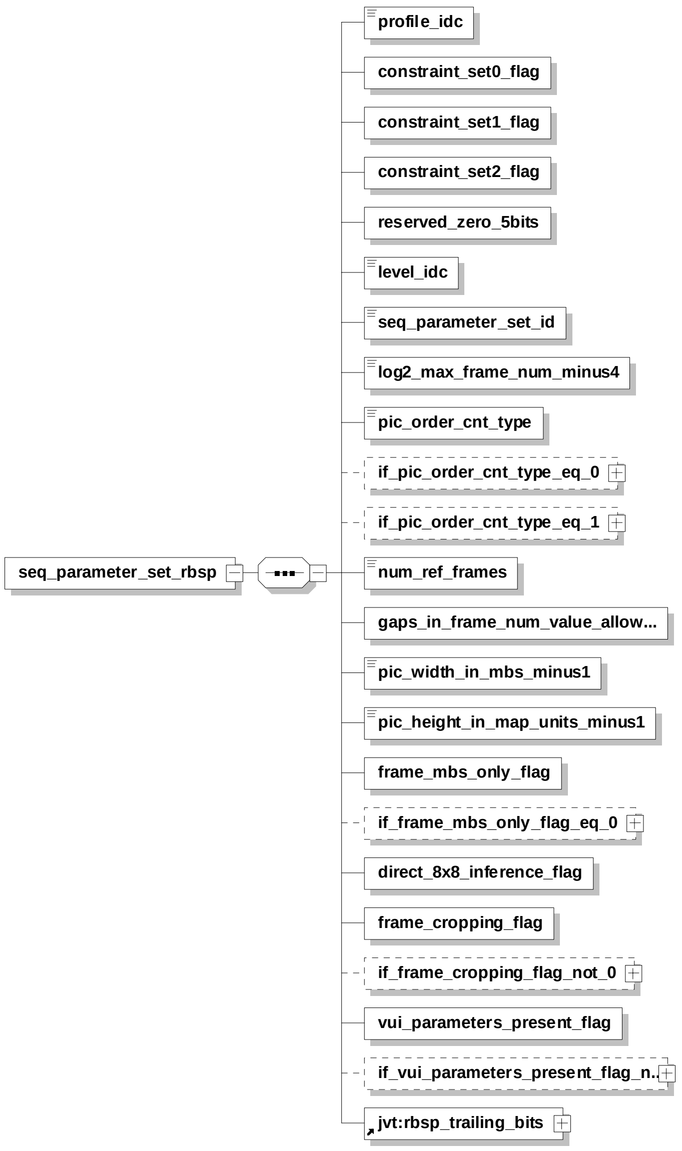

seq_parameter_set_rbsp
profile_idc
constraint_set0_flag
constraint_set1_flag
constraint_set2_flag
reserved_zero_5bits
level_idc
seq_parameter_set_id
log2_max_frame_num_minus4
pic_order_cnt_type
if_pic_order_cnt_type_eq_0
if_pic_order_cnt_type_eq_1
num_ref_frames
gaps_in_frame_num_value_allow...
pic_width_in_mbs_minus1
pic_height_in_map_units_minus1
frame_mbs_only_flag
if_frame_mbs_only_flag_eq_0
direct_8x8_inference_flag
frame_cropping_flag
if_frame_cropping_flag_not_0
vui_parameters_present_flag
if_vui_parameters_present_flag_n..
jvt:rbsp_trailing_bits

Figure B.10: BS Schema for the SPS syntax of H.264/AVC.

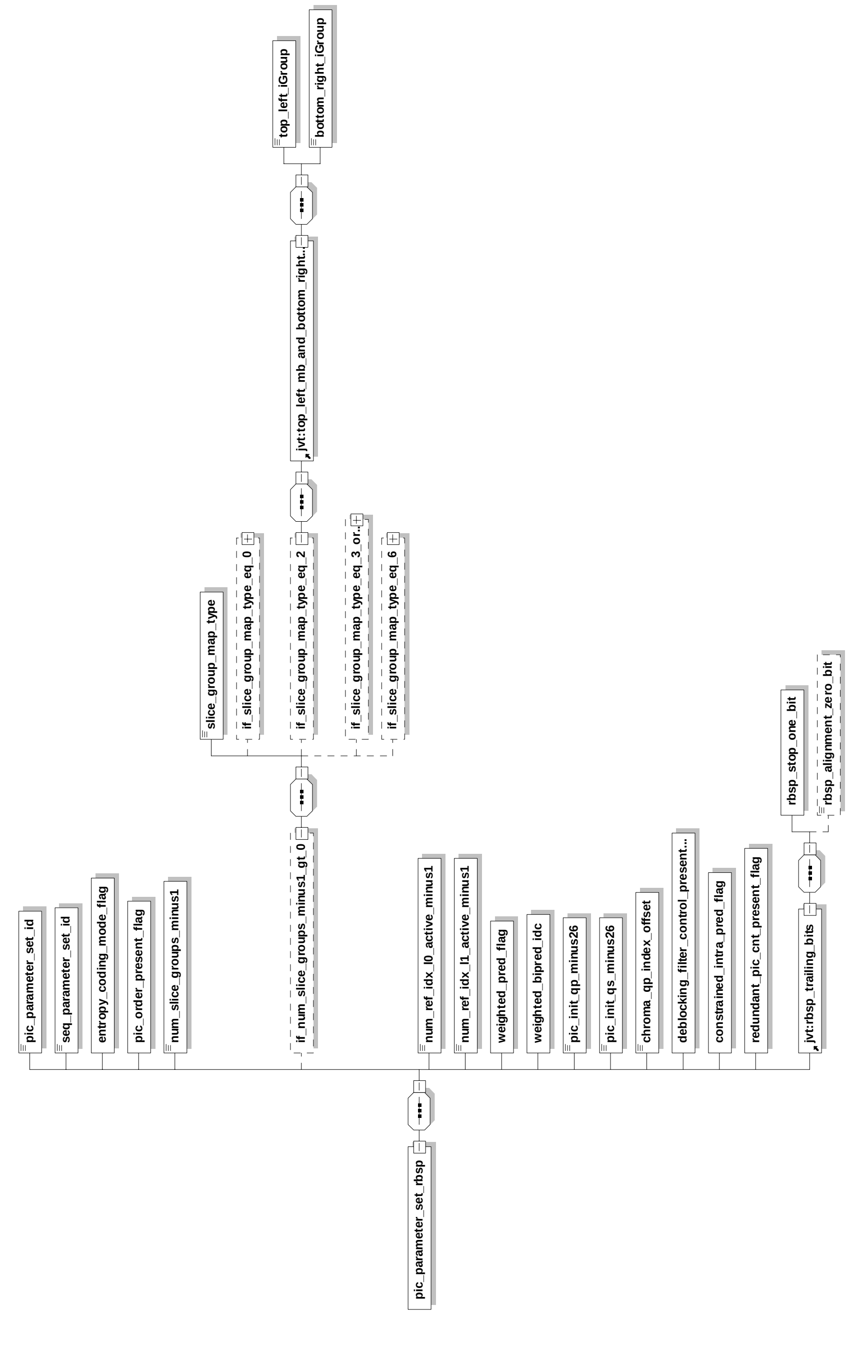


Figure B.11: BS Schema for the PPS syntax of H.264/AVC.

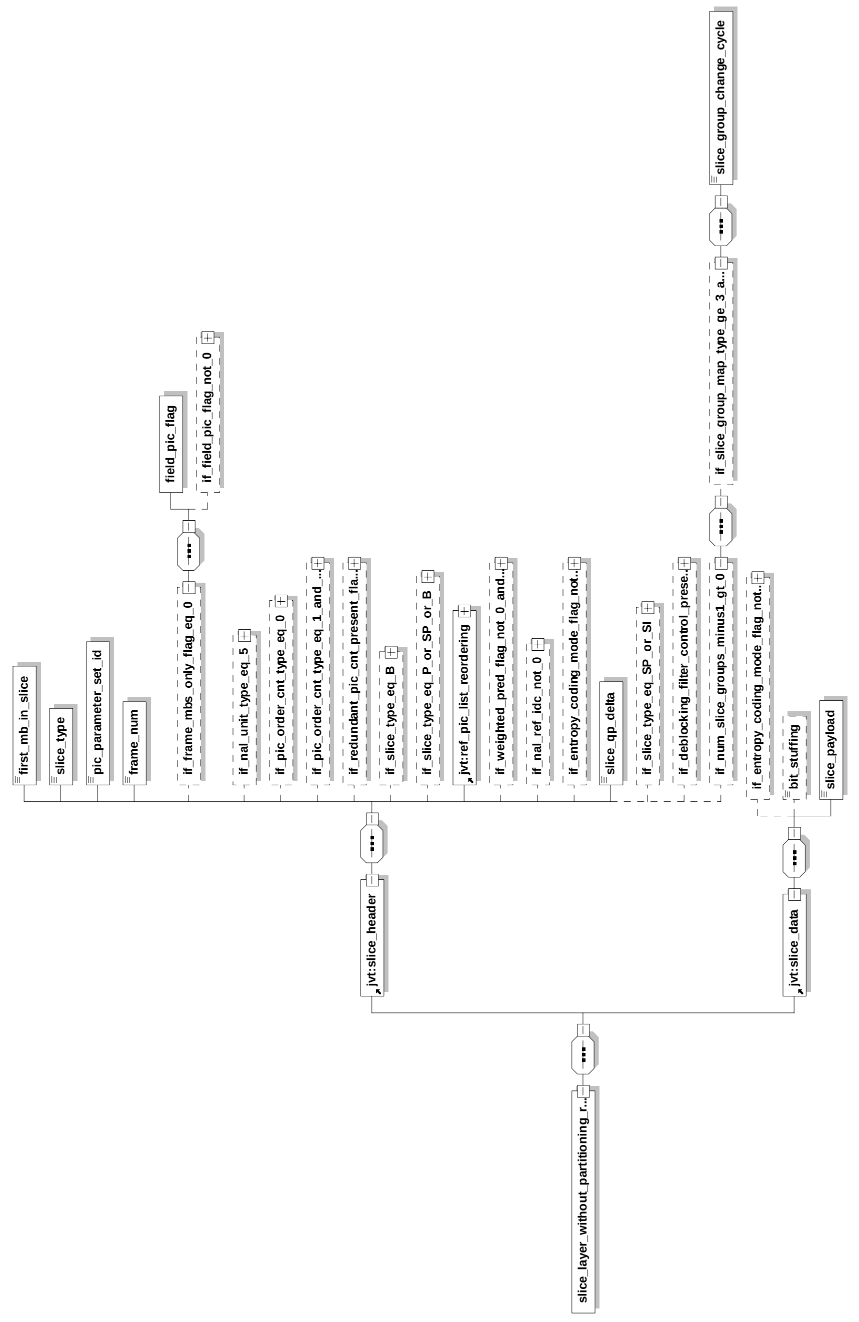

slice_layer_without_partitioning_r...
jvt:slice_header
jvt:slice_data
first_mb_in_slice
slice_type
pic_parameter_set_id
frame_num
if_frame_mbs_only_flag_eq_0
field_pic_flag
if_field_pic_flag_not_0
if_nal_unit_type_eq_5
if_pic_order_cnt_type_eq_0
if_pic_order_cnt_type_eq_1_and_...
if_redundant_pic_cnt_present_fla...
if_slice_type_eq_B
if_slice_type_eq_P_or_SP_or_B
jvt:ref_pic_list_reordering
if_weighted_pred_flag_not_0_and...
if_nal_ref_idc_not_0
if_entropy_coding_mode_flag_not...
slice_qp_delta
if_slice_type_eq_SP_or_SI
if_deblocking_filter_control_prese...
if_num_slice_groups_minus1_gt_0
if_slice_group_map_type_ge_3_a...
slice_group_change_cycle
if_entropy_coding_mode_flag_not...
bit_stuffing
slice_payload

Figure B.12: BS Schema for the slice syntax of H.264/AVC.

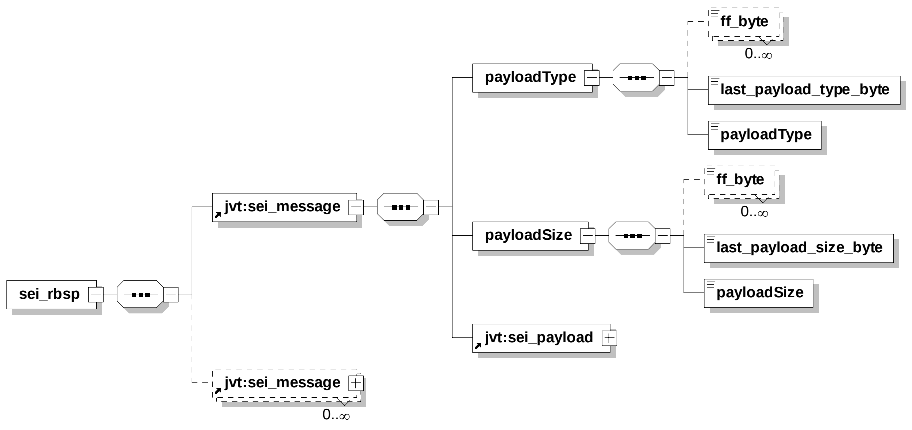


Figure B.13: BS Schema for the high-level SEI syntax of H.264/AVC.

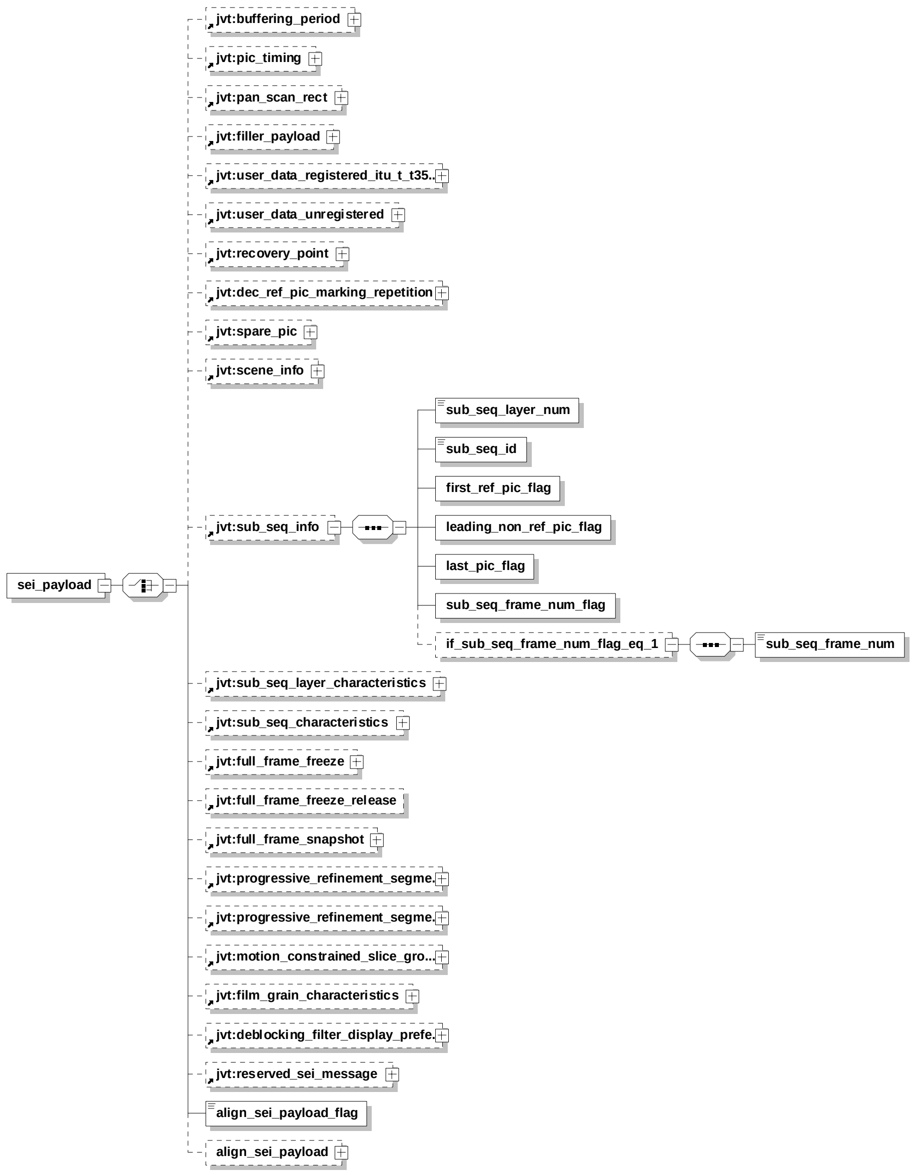

jvt:buffering_period
jvt:pic_timing
jvt:pan_scan_rect
jvt:filler_payload
jvt:user_data_registered_itu_t_t35..
jvt:user_data_unregistered
jvt:recovery_point
jvt:dec_ref_pic_marking_repetition
jvt:spare_pic
jvt:scene_info
sub_seq_layer_num
sub_seq_id
first_ref_pic_flag
jvt:sub_seq_info
leading_non_ref_pic_flag
sei_payload
last_pic_flag
sub_seq_frame_num_flag
if_sub_seq_frame_num_flag_eq_1
sub_seq_frame_num
jvt:sub_seq_layer_characteristics
jvt:sub_seq_characteristics
jvt:full_frame_freeze
jvt:full_frame_freeze_release
jvt:full_frame_snapshot
jvt:progressive_refinement_segme..
jvt:progressive_refinement_segme..
jvt:motion_constrained_slice_gro...
jvt:film_grain_characteristics
jvt:deblocking_filter_display_prefe..
jvt:reserved_sei_message
align_sei_payload_flag
align_sei_payload

Figure B.14: BS Schema fragment containing the different SEI messages that can be used in H.264/AVC (including the SEI messages specified by the FRExt amendment).

Appendix C

Context management for BSDL's BintoBSD

C.1 Introduction

BSDL's BintoBSD process is responsible for translating the high-level structure of a binary media resource into an XML description. This description can then be used for further processing. However, as discussed in Chapter 4 and Chapter 5, this format-agnostic translation process is characterized by an increasing memory usage and a decreasing processing speed in terms of the number of parse units processed.

One solution for the performance issues of the BintoBSD process, which can be considered fundamental in the context of video bitstreams, is to enhance the MPEG-21 BSDL standard. The development of such extensions was addressed by the Multimedia Lab research group, resulting in the definition of five additional attributes on top of the initial version of the BSDL specification[1]. These attributes are discussed in more detail in the next section.

C.2 BSD generation using context management

The in-memory representation of a BSD, which is commonly referred to as the context, is needed by the BintoBSD process for the correct evaluation of an arbitrary set of XPath 1.0 expressions. These XPath expressions, residing in a BS Schema, for instance provide support for features such as conditional parsing and loop constructs.

[1]As part of the first version of the MPEG-21 DIA standard.

Four new attributes are introduced in BSDL to keep the context minimal during the execution of the BintoBSD process, while still guaranteeing a correct evaluation of the different XPath expressions embedded in a BS Schema (on the condition that the attributes are correctly used by a BS Schema author). These four attributes are `bs0:startContext`, `bs0:stopContext`, `bs0:partContext`, and `bs0:redefineMarker`; they can be used by a BS Schema author to prune the parts of the in-memory tree representation of a BSD that are no longer required for the evaluation of XPath expressions.

A fifth new attribute, named `bs0:defaultTreeInMemory`, is defined for the purpose of compatibility with already existing BS Schemata and implementations of the BintoBSD process.

A brief explanation of the semantics of the five newly defined attributes is provided below. Their use is illustrated with an example presented further in this section.

- `bs0:startContext`: this attribute signals to a BintoBSD Parser that the element, to which the attribute belongs, is needed to evaluate a forthcoming XPath expression. Consequently, a BintoBSD Parser will add this element to the in-memory representation of the BSD. In a later phase of the BSD generation process, a unique marker can be used to remove the element from the context. This marker is provided as a value for the `bs0:startContext` attribute.

- `bs0:stopContext`: this attribute is the counterpart of the `bs0:startContext` attribute: it allows to reduce the size of the in-memory representation of the BSD. Indeed, this attribute contains a list of markers, which denote the elements that have to be removed from the context.

- `bs0:partContext`: the purpose of this attribute is similar to the purpose of the `bs0:startContext` attribute: both attributes contribute to the size of the context. However, in contrast to `bs0:startContext`, it does not take a marker as value.

  Indeed, the `bs0:partContext` attribute can only have one out of two possible values: `true` or `false`. A value of `true` implies that the element, to which the attribute belongs, needs to be added to the context. A value of `false`, which is the default value for this attribute in the case it is missing for a particular element, signals to a BintoBSD Parser that it is not necessary to add the corresponding element to the in-memory representation of a BSD.

As `bs0:partContext` does not have a marker as value, it is not possible to explicitly remove the element it is associated with from the context. This can only be achieved by the removal of an ancestor element using `bs0:stopContext`.

The `bs0:partContext` attribute will typically be used when its associated element is needed in the location step of an XPath expression. As such, this attribute allows to achieve a limited memory consumption without having to employ an abundant number of markers.

- `bs0:redefineMarker`: this attribute can be used to rename an already existing marker, giving the author of a BS Schema advanced control pertaining to the in-memory representation of a BSD.
- `bs0:defaultTreeInMemory`: this attributes can have one out of two possible values: `true` or `false`. A value of `false` means that the context management attributes are in use - by default, nothing will be added to the context by a BintoBSD Parser, unless otherwise instructed by the context management attributes. A value of `true` means that the entire BSD needs to be stored in memory.

To test the expressive power of the context management attributes, they were employed in the different BS Schemata discussed in Annex B. Their use is also illustrated in Listing C.1, which contains a further development of the generic BS Schema template that was introduced in Chapter 4. The listing shows that the context management is typically done at the level of the fundamental unit of parsing for a BintoBSD Parser (e.g., a NAL unit for H.264/AVC, an EBDU for VC-1, et cetera).

Listing C.1: Generic BS Schema for a packet-based media format, annotated with context management attributes.

```
<xsd:schema bs0:defaultTreeInMemory="false">
  <xsd:element name="bitstream" bs0:startContext="'bitstream'">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parse_unit"
           bs0:startContext="'pu'" bs0:stopContext="'old_pu'">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="coded_header_data"/>
              <xsd:element ref="coded_media_data"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="coded_header_data"
     bs0:redefineMarker="'header_pu' 'old_pu' 'pu' 'header_pu'">
    <!-- ... -->
  </xsd:element>
  <xsd:element name="coded_media_data"
     bs0:redefineMarker="'data_pu' 'old_pu' 'pu' 'data_pu'">
    <!-- ... -->
  </xsd:element>
</xsd:schema>
```

Figure C.1 contains the typical evolution of the tree-based in-memory representation of a BSD when using the context management approach proposed in Listing C.1. The boxes represent nodes, while the labels outside the boxes represent markers.

1. Layout of the context after having processed a first parse unit conveying coded header data.

2. Layout of the context after having processed a second parse unit containing coded media data.

3. Layout of the context after having processed a third parse unit carrying coded media data. Note how the marker for the first unit containing media data is renamed from `data_pu` to `old_pu`. This signals to a BintoBSD Parser that this data unit may be removed from the context when processing a next parse unit.

4. Layout of the context after having processed a fourth parse unit transporting coded media data. The dashed lines denote a node that is removed from the context.

For a more extensive overview regarding the development and use of the context management attributes, we would like to refer the interested reader to [21,28,32,35]. A preliminary performance assessment regarding the use of the context management attributes, compared to the use of BFlavor for BSD generation for H.264/AVC bitstreams, can be found in Chapter 6. Similar performance results can be found in [28] for bitstreams that are compliant with VC-1.

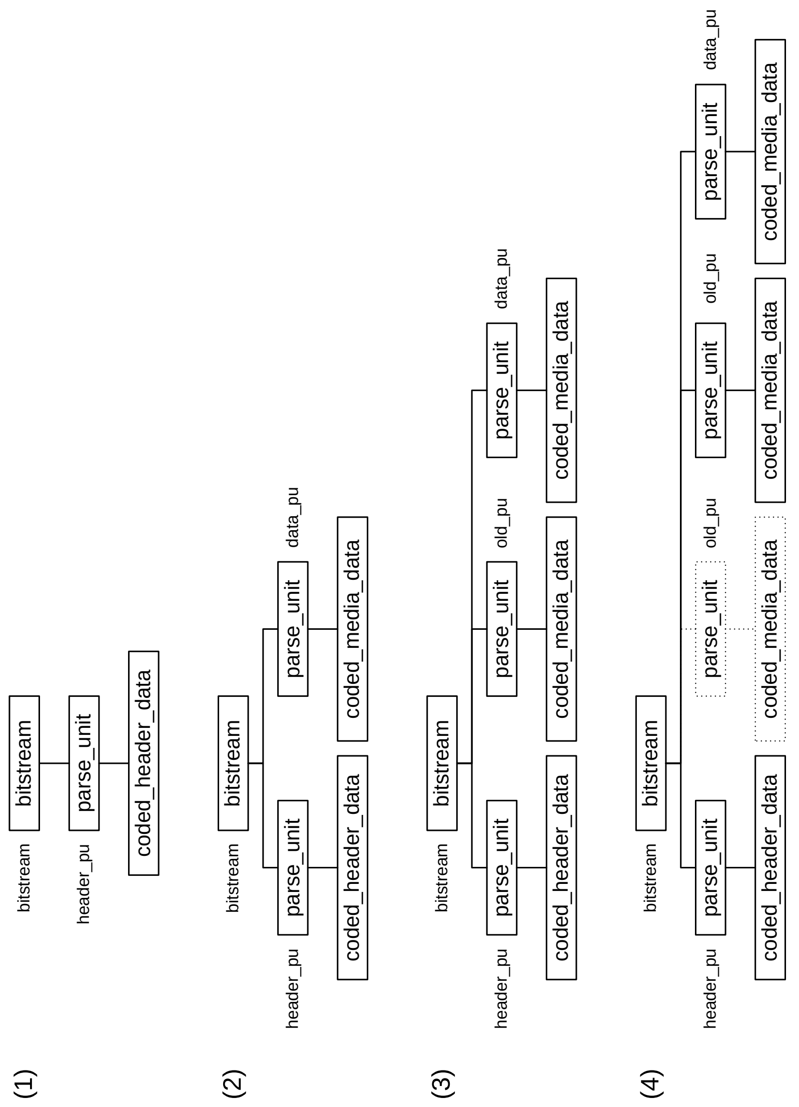


Figure C.1: Typical evolution of the context when using context management attributes.

Appendix D

Stylesheets for BSD transformation

D.1 Introduction

In the context of this research, several stylesheets have been developed for the purpose of transforming or manipulating XML-based BSDs. A number of these stylesheets were initially written using XSLT. However, XSLT cannot be used in practice for the manipulation of large BSDs (i.e., BSDs with a size typically larger than 50 MB) due to its memory-consuming nature. This became particularly visible once the performance problems of BSDL's BintoBSD Parser were circumvented or solved, allowing the processing of larger elementary video bitstreams, and hence allowing the creation of verbose BSDs. As such, the more efficient generation of BSDs resulted in a shift of the performance bottleneck in the BSD-driven content adaptation chain from the BSD creation process, either using BFlavor or an optimized BintoBSD Parser, to the BSD transformation step, using XSLT.

A first attempt to obtain a more efficient BSD transformation process consisted of the employment of a hybrid transformation approach, based on the combined use of STX and XSLT logic in a single STX stylesheet (see for instance Listing D.5 in the next section). The code written in STX is used for iterating through the descriptions of the different parse units in a BSD and for passing them to the XSLT logic. This XSLT code is subsequently used for performing a more detailed analysis of the description of a parse unit. For instance, in case the XSLT logic detects a description of a bidirectionally coded picture, it might decide to remove or adapt this description.

The hybrid approach, as described above, makes it possible to acquire a low memory footprint during the processing of a BSD, as well as a straightforward reuse of the logic that was previously written in XSLT. However, for our purposes, this work flow was not able to operate in real time when making use of the Joost STX engine, which was, at the time of writing, the only engine available for the execution of STX stylesheets. The inefficient performance behavior is due to the fact that XSLT code is to be executed for every parse unit described in a BSD. This implies that the STX run-time needs to initialize and call an XSLT engine for every parse unit. For instance, in order to transform a BSD in real time for an H.264/AVC bitstream that is characterized by a throughput of 120 NAL units per second, the STX engine needs to be able to call an XSLT engine 120 times per second.

A second attempt to transform BSDs in a more efficient way, consisted of the use of stylesheets that were entirely written in STX[1]. This approach effectively allows a memory-efficient, streaming-based transformation of verbose BSDs in real time.

A number of relevant and representative stylesheets are listed in the next section of this appendix. The structure of the stylesheets is in line with the generic BS Schema template presented in Listing 4.10 in Chapter 4. Note that the BSD transformations can also be implemented by using software that is written on top of a DOM or SAX API. However, this approach was not followed since MPEG-21 DIA envisions a generic content adaptation framework that is entirely steered by XML technologies. However, it is noteworthy that SAX filters can be easily updated in order to reflect changes in an adaptive multimedia framework (e.g., to anticipate a varying bandwidth). Such functionality is for example necessary for the purpose of dynamic content adaptation. The use of SAX filters is in contrast to the use of stylesheets, which are characterized by a rather static nature: their behavior cannot easily be changed in a dynamic way. For a more detailed overview regarding the efficiency of different BSD transformation technologies, we would like to refer the interested reader to [16], [27], [28], and [38].

D.2 XSLT, STX, and XSLT/STX stylesheets

Listing D.1: XSLT stylesheet for adding SEI messages to an elementary H.264/AVC bitstream with a coding structure as visualized in Figure 3.9.

```
<xsl:stylesheet xmlns:jvt="h264_avc">
  <xsl:output method="xml" indent="yes"/>
  <!-- Match all. -->
```

[1]The STX specification could not be referenced normatively from the DIA standard as this document is not a standard. This is in contrast to XSLT, which is a W3C recommendation.

```
<xsl:template name="tplAll" match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
<!-- Do nothing for processing instructions. -->
<xsl:template match="processing-instruction()"/>
<xsl:template name="set_gap_in_frame_num"
     match="jvt:byte_stream_nal_unit_sps/jvt:nal_unit_sps/
            jvt:raw_byte_sequence_payload_sps/
            jvt:seq_parameter_set_rbsp/
            jvt:gaps_in_frame_num_value_allowed_flag">
  <!-- Adjust gaps_in_frame_num_value_allowed_flag. -->
  <gaps_in_frame_num_value_allowed_flag>
    1
  </gaps_in_frame_num_value_allowed_flag>
</xsl:template>
<!-- Template for adding SEI messages to the IDR pictures in
     the base layer. -->
<xsl:template name="layer0_IDR"
     match="jvt:byte_stream_nal_unit[jvt:nal_unit[
            jvt:raw_byte_sequence_payload[
            jvt:coded_slice_of_an_IDR_picture[
            jvt:slice_layer_without_partitioning_rbsp
            [jvt:slice_header
            [jvt:first_mb_in_slice = 0 and
            jvt:slice_type=7]]]]]]">

  <byte_stream_nal_unit>
    <zero_byte>0</zero_byte>
    <start_code_prefix_one_3bytes>
      000001
    </start_code_prefix_one_3bytes>
    <nal_unit>
      <forbidden_zero_bit>0</forbidden_zero_bit>
      <nal_ref_idc>0</nal_ref_idc>
      <nal_unit_type>6</nal_unit_type>
      <raw_byte_sequence_payload>
        <sei_rbsp>
          <sei_message>
            <last_payload_type_byte>
              10
            </last_payload_type_byte>
            <last_payload_size_byte>
              1
            </last_payload_size_byte>
            <sei_payload>
              <sub_seq_info>
                <sub_seq_layer_num>0</sub_seq_layer_num>
```

```
                  <sub_seq_id>0</sub_seq_id>
                  <!-- The values below are not correct. -->
                  <first_ref_pic_flag>0</first_ref_pic_flag>
                  <leading_non_ref_pic_flag>
                    0
                  </leading_non_ref_pic_flag>
                  <last_pic_flag>0</last_pic_flag>
                  <sub_seq_frame_num_flag>
                    0
                  </sub_seq_frame_num_flag>
                </sub_seq_info>
              </sei_payload>
            </sei_message>
            <rbsp_trailing_bits>
              <rbsp_stop_one_bit>1</rbsp_stop_one_bit>
              <rbsp_alignment_zero_bit>
                0
              </rbsp_alignment_zero_bit>
            </rbsp_trailing_bits>
          </sei_rbsp>
        </raw_byte_sequence_payload>
      </nal_unit>
    </byte_stream_nal_unit>
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <!-- Template for adding SEI messages to the P pictures
       in the base layer. -->
  <!-- Template for adding SEI messages to the B pictures
       in the first enhancement layer. -->
  <xsl:template name="layer1"
       match="jvt:byte_stream_nal_unit[jvt:nal_unit[
              jvt:raw_byte_sequence_payload[
              jvt:coded_slice_of_a_non_IDR_picture[
              jvt:slice_layer_without_partitioning_rbsp[
              jvt:slice_header
              [jvt:first_mb_in_slice = 0 and
              jvt:frame_num mod 4 = 2]]]]]]">
    <byte_stream_nal_unit>
      <zero_byte>0</zero_byte>
            <start_code_prefix_one_3bytes>
        000001
      </start_code_prefix_one_3bytes>
      <nal_unit>
        <forbidden_zero_bit>0</forbidden_zero_bit>
        <nal_ref_idc>0</nal_ref_idc>
        <nal_unit_type>6</nal_unit_type>
        <raw_byte_sequence_payload>
```

```
          <sei_rbsp>
            <sei_message>
              <last_payload_type_byte>
                10
              </last_payload_type_byte>
              <last_payload_size_byte>
                1
              </last_payload_size_byte>
              <sei_payload>
                <sub_seq_info>
                  <sub_seq_layer_num>1</sub_seq_layer_num>
                  <xsl:element name="sub_seq_id">
                    <xsl:value-of select="floor(jvt:nal_unit/
                     jvt:raw_byte_sequence_payload/
                     jvt:coded_slice_of_a_non_IDR_picture/
                     jvt:slice_layer_without_partitioning_rbsp/
                     jvt:slice_header/jvt:frame_num div 4)"/>
                  </xsl:element>
                  <!-- The values below are not correct. -->
                  <first_ref_pic_flag>0</first_ref_pic_flag>
                  <leading_non_ref_pic_flag>
                    0
                  </leading_non_ref_pic_flag>
                  <last_pic_flag>0</last_pic_flag>
                  <sub_seq_frame_num_flag>
                    0
                  </sub_seq_frame_num_flag>
                </sub_seq_info>
              </sei_payload>
            </sei_message>
            <rbsp_trailing_bits>
              <rbsp_stop_one_bit>1</rbsp_stop_one_bit>
              <rbsp_alignment_zero_bit>
                0
              </rbsp_alignment_zero_bit>
            </rbsp_trailing_bits>
          </sei_rbsp>
        </raw_byte_sequence_payload>
      </nal_unit>
    </byte_stream_nal_unit>
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <!-- Template for adding SEI messages to the B pictures
       in the second enhancement layer. -->
  <!-- Template for adding SEI messages to the B pictures
       in the third enhancement layer. -->
</xsl:stylesheet>
```

Listing D.2: XSLT stylesheet for the SEI-based disposal of the second and third temporal enhancement layer in an elementary H.264/AVC bitstream, having a coding structure as visualized in Figure 3.9.

```
<xsl:stylesheet xmlns:jvt="h264_avc">
  <xsl:output method="xml" indent="yes"/>
  <!-- Match all. -->
  <xsl:template name="tplAll" match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <!-- Do nothing for processing instructions. -->
  <xsl:template match="processing-instruction()"/>
  <!-- Template for deleting pictures. -->
  <xsl:template name="deletePicture"
                match="jvt:byte_stream_nal_unit
                [jvt:nal_unit[jvt:nal_unit_type != 6]]">
          <xsl:if test="preceding-sibling::
                  jvt:byte_stream_nal_unit
                        [position() &lt;= 3]
                        /jvt:nal_unit/
                        jvt:raw_byte_sequence_payload/
                        jvt:sei_rbsp/jvt:sei_message/
                        jvt:sei_payload/jvt:sub_seq_info/
                        jvt:sub_seq_layer_num &lt;= 1">
            <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
      </xsl:copy>
    </xsl:if>
  </xsl:template>
  <!-- Template for deleting redundant SEI messages. -->
  <xsl:template name="deleteSEI"
                match="jvt:byte_stream_nal_unit[
                jvt:nal_unit[jvt:raw_byte_sequence_payload[
                jvt:sei_rbsp[jvt:sei_message[jvt:sei_payload[
                jvt:sub_seq_info[
                jvt:sub_seq_layer_num &gt;= 2]]]]]]]"/>
</xsl:stylesheet>
```

Listing D.3: STX stylesheet for the removal of the second and third temporal enhancement layer in an elementary H.264/AVC bitstream, having a coding structure as visualized in Figure 3.9.

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:sf="http://stx.sourceforge.net/2003/functions"
     pass-through="all" output-method="xml"
```

```
   xmlns:jvt="h264_avc" strip-space="yes">
<!-- Declaration of global variables. -->
<stx:variable name="copy_nal_unit" select="false()"/>
<stx:variable name="slice_type" select="0"/>
<!-- Declaration of buffers. -->
<stx:buffer name="nal_unit"/>
<!-- Put a complete NAL unit in a buffer, modify this NAL
     unit if needed, and eventually copy the result to the
     output stream. -->
<stx:template match="jvt:byte_stream_nal_unit">
  <stx:result-buffer name="nal_unit" clear="yes">
    <!-- Copy everything from the source stream into the
         buffer. -->
    <stx:copy>
      <!-- Process other templates. -->
      <stx:process-children/>
    </stx:copy>
  </stx:result-buffer>
  <stx:if test="$copy_nal_unit">
    <stx:process-buffer group="empty" name="nal_unit"/>
  </stx:if>
</stx:template>
<!-- Adjust gaps_in_frame_num_value_allowed_flag. -->
<stx:template
     match="jvt:gaps_in_frame_num_value_allowed_flag">
  <stx:element name="gaps_in_frame_num_value_allowed_flag"
               namespace="h264_avc">1</stx:element>
</stx:template>
<!-- Check nal_ref_idc; take appropriate action. -->
<stx:template match="jvt:nal_unit_type">
  <stx:assign name="copy_nal_unit" select="true()"/>
  <stx:copy>
    <stx:value-of select="."/>
  </stx:copy>
  <stx:if test=". != 7">
    <stx:process-siblings group="removeLayer_2_and_3"/>
  </stx:if>
</stx:template>
<!-- Check whether a slice has to be kept or needs to be
     thrown away. -->
<stx:group name="removeLayer_2_and_3">
  <stx:template match="jvt:slice_type" public="no">
    <stx:copy>
      <stx:value-of select="."/>
    </stx:copy>
    <stx:assign name="slice_type" select="."/>
  </stx:template>
  <stx:template match="jvt:frame_num" public="no">
    <stx:copy>
```

```
        <stx:value-of select="."/>
      </stx:copy>
      <!-- Check whether the current slice belongs to the
           second or the third enhancement layer. -->
      <stx:if test="($slice_type = 1 or $slice_type = 6) and
                    (. mod 4 != 2)">
        <stx:assign name="copy_nal_unit" select="false()"/>
      </stx:if>
    </stx:template>
  </stx:group>
  <!-- The empty group, which is used to copy something to
       the output stream. -->
  <stx:group name="empty"/>
</stx:transform>
```

Listing D.4: STX stylesheet for replacing B slices by skipped P slices in an elementary H.264/AVC bitstream.

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:sf="http://stx.sourceforge.net/2003/functions"
     pass-through="all" output-method="xml"
     xmlns:jvt="h264_avc" strip-space="yes">
  <!-- Declaration of global variables. -->
  <stx:variable name="copy_nal_unit" select="false()"/>
  <stx:variable name="nal_ref_idc" select="0"/>
  <stx:variable name="first_mb_in_slice" select="0"/>
  <stx:variable name="slice_type" select="-1"/>
  <!-- Declaration of buffers. -->
  <stx:buffer name="nal_unit"/>
  <stx:buffer name="payload"/>
  <!-- Put a complete NAL unit in a buffer, modify this NAL
       unit if needed, and eventually copy the result to the
       output stream. -->
  <stx:template match="jvt:byte_stream_nal_unit">
    <stx:result-buffer name="nal_unit" clear="yes">
      <!-- Copy everything from the source stream into the
           buffer. -->
      <stx:copy>
        <!-- Process other templates. -->
        <stx:process-children/>
      </stx:copy>
    </stx:result-buffer>
    <stx:process-buffer group="empty" name="nal_unit"/>
  </stx:template>
        <!-- Check nal_ref_idc; take appropriate action. -->
  <stx:template match="jvt:nal_ref_idc">
    <stx:assign name="slice_type" select="-1"/>
    <stx:assign name="nal_ref_idc" select="."/>
    <stx:copy>
      <stx:value-of select="."/>
    </stx:copy>
    <stx:if test=". = 0">
      <stx:process-siblings group="nonReference"/>
    </stx:if>
  </stx:template>
  <!-- Group that processes non-reference NAL units. -->
  <stx:group name="nonReference">
    <stx:template match="jvt:raw_byte_sequence_payload"
                  public="no">
      <!-- Keep the payload in a separate buffer for further
           processing. -->
      <stx:result-buffer name="payload" clear="yes">
        <stx:copy>
          <stx:process-children group="empty"/>
        </stx:copy>
      </stx:result-buffer>
      <!-- Retrieve relevant information from the payload. -->
      <stx:process-buffer name="payload"
                          group="parse_payload_buffer"/>
      <stx:if test="$slice_type = 1 or $slice_type = 6">
        <stx:process-buffer name="payload"
                            group="BtoskippedP"/>
      </stx:if>
      <stx:else>
        <stx:process-buffer name="payload" group="empty"/>
      </stx:else>
    </stx:template>
  </stx:group>
  <!-- Group that retrieves relevant information from the
       payload of a NAL unit. -->
  <stx:group name="parse_payload_buffer" pass-through="none">
    <!-- Remember the macroblock number of the first macroblock
         of the slice. -->
    <stx:template match="jvt:first_mb_in_slice" public="no">
      <stx:assign name="first_mb_in_slice" select="."/>
    </stx:template>
    <!-- Remember the slice type of the current slice. -->
    <stx:template match="jvt:slice_type" public="no">
      <stx:assign name="slice_type" select="."/>
    </stx:template>
  </stx:group>
  <!-- Group that translates the syntax of B slices into
       skipped P slices. -->
  <stx:group name="BtoskippedP">
    <!-- Two templates to rename syntax elements (note the
         "process-children" construction). -->
```

```
<stx:template match="jvt:coded_slice_of_a_non_IDR_picture"
              public="no">
  <stx:element
       name="coded_slice_of_a_skipped_non_IDR_picture"
       namespace="h264_avc">
    <stx:process-children group="BtoskippedP"/>
  </stx:element>
</stx:template>
<stx:template
     match="jvt:slice_layer_without_partitioning_rbsp"
     public="no">
  <stx:element
       name="skipped_slice_layer_without_partitioning_rbsp"
       namespace="h264_avc">
    <stx:process-children group="BtoskippedP"/>
  </stx:element>
</stx:template>
<!-- Change the slice type. -->
<stx:template match="jvt:slice_type" public="no">
  <stx:element name="slice_type" namespace="h264_avc">
    0
  </stx:element>
</stx:template>
<!-- Change the slice_qp_delta: this saves a number of
     bits. -->
<stx:template match="jvt:slice_qp_delta" public="no">
  <stx:element name="slice_qp_delta" namespace="h264_avc">
    0
  </stx:element>
</stx:template>
<!-- Every syntax element regarding B slices can be
     dropped. -->
<stx:template match="jvt:if_slice_type_eq_B" public="no">
</stx:template>
<!-- Replace the coded slice data with skipped slice
     data. -->
<stx:template match="jvt:slice_data" public="no">
  <stx:element name="skipped_slice_data"
               namespace="h264_avc">
    <stx:if test="$first_mb_in_slice = 0 or
                  $first_mb_in_slice = 233 or
                  $first_mb_in_slice = 466 or
                  $first_mb_in_slice = 699">
      <stx:element name="mb_skip_run" namespace="h264_avc">
        233
      </stx:element>
    </stx:if>
    <stx:else>
      <stx:element name="mb_skip_run" namespace="h264_avc">
```

```
          234
        </stx:element>
      </stx:else>
      <stx:element name="rbsp_trailing_bits"
                   namespace="h264_avc">
        <stx:element name="rbsp_stop_one_bit"
                     namespace="h264_avc">
          <stx:value-of select="1"/>
        </stx:element>
        <stx:element name="rbsp_alignment_zero_bit"
                     namespace="h264_avc">
          <stx:value-of select="0"/>
        </stx:element>
      </stx:element>
    </stx:element>
  </stx:template>
</stx:group>
<!-- The empty group, which is used to copy something to
     the output stream. -->
<stx:group name="empty"/>
</stx:transform>
```

Listing D.5: A hybrid STX stylesheet for the removal of non-reference B slice coded pictures in an elementary H.264/AVC bitstream.

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:jvt="h264_avc" xmlns="h264_avc"
           strip-space="yes" output-method="xml">
        <!-- Declaration of variables. -->
        <stx:variable name="xslt"
             select="'http://www.w3.org/1999/XSL/Transform'"/>
  <!-- Declaration of a buffer that contains XSLT code. -->
  <stx:buffer name="xslt_code_buffer">
    <xsl:stylesheet
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:jvt = "h264_avc" xmlns = "h264_avc">
      <xsl:output method="xml" indent="yes"/>
      <!-- Default template: the identity transformation. -->
      <xsl:template name="tplAll" match="@*|node()">
        <xsl:copy>
          <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
      </xsl:template>
      <!-- Check whether the current slice is a non-reference
           b slice. -->
      <xsl:template name="tplSlice"
           match="jvt:byte_stream_nal_unit[
            jvt:nal_unit[
            jvt:nal_ref_idc = 0 and
            jvt:raw_byte_sequence_payload[
            jvt:coded_slice_of_a_non_IDR_picture[
            jvt:slice_layer_without_partitioning_rbsp[
            jvt:slice_header[jvt:slice_type = 6]]]]]]"/>
    </xsl:stylesheet>
  </stx:buffer>
  <!-- STX template, implementing the identity
       transformation. -->
  <stx:template match="@*|node()">
    <stx:copy>
      <stx:process-children/>
    </stx:copy>
  </stx:template>
  <!-- STX template, responsible for calling the embedded
       XSLT stylesheet. -->
  <stx:template match="jvt:byte_stream_nal_unit">
    <stx:if test="filter-available($xslt)">
      <stx:process-self filter-method="{$xslt}"
           filter-src="buffer(xslt_code_buffer)"/>
    </stx:if>
    <stx:else>
      <stx:message>
        Cannot invoke an XSLT transformation.
      </stx:message>
    </stx:else>
  </stx:template
</stx:transform>
```

Listing D.6: A hybrid XSLT/STX stylesheet for the removal of audio packets in an MPEG-2 Program Stream (stored in a private stream).

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:mp2="MPEG2"
     strip-space="yes"
     output-method="xml">
  <!-- Declaration of variables. -->
  <stx:variable name="pack_count" select="0"/>
  <stx:variable name="xslt"
       select="'http://www.w3.org/1999/XSL/Transform'"/>
  <!-- Declaration of a buffer that contains XSLT code. -->
  <stx:buffer name="xslt_code_buffer">
    <xsl:stylesheet>
```

```
      <xsl:template name="tplAll" match="node()">
        <xsl:copy>
          <xsl:apply-templates select="node()"/>
        </xsl:copy>
      </xsl:template>
      <!-- Check whether audio or padding streams
           are available. -->
      <xsl:template match="mp2:pack[
           count(mp2:PES_packet/mp2:private_stream_1) &gt; 0 or
           count(mp2:PES_packet/mp2:private_stream_2)]">
        <!-- Do nothing. -->
      </xsl:template>
    </xsl:stylesheet>
  </stx:buffer>
  <!-- Default STX template. -->
  <stx:template match="@*|node()">
    <stx:copy>
      <stx:process-children/>
    </stx:copy>
  </stx:template>
  <!-- An XSLT stylesheet is applied for every pack. -->
  <stx:template match="mp2:pack">
    <stx:if test="filter-available($xslt)">
      <stx:process-self filter-method="{$xslt}"
                        filter-src="buffer(xslt_code_buffer)"/>
      <stx:assign name="pack_count" select="$pack_count + 1"/>
      <stx:message>
        Pack <stx:value-of select="$pack_count"/> processed.
      </stx:message>
    </stx:if>
    <stx:else>
      <stx:message>
        Cannot invoke an XSLT transformation.
      </stx:message>
    </stx:else>
  </stx:template>
</stx:transform>
```

Listing D.7: A STX stylesheet for the extraction of an elementary video bitstream from an MPEG-2 Systems Program Stream (only containing PES packets with video data).

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:mp2="MPEG2"
     strip-space="yes"
     output-method="xml">
  <!-- Declaration of variables. -->
```

```
<stx:variable name="pack_count" select="0"/>
 <!-- Templates. -->
<stx:template match="@*|node()">
  <stx:copy>
    <stx:process-children/>
  </stx:copy>
</stx:template>
<stx:template match="mp2:pack/mp2:pack_header">
  <stx:assign name="pack_count" select="$pack_count + 1"/>
  <stx:message>
    Pack <stx:value-of select="$pack_count"/> processed.
  </stx:message>
  <!-- Do nothing. -->
</stx:template>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:packet_start_code_prefix"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:stream_id"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_packet_length"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:filler"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_scrambling_control"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_priority"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:data_alignment_indicator"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:copyright"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:original_or_copy"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PTS_DTS_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:ESCR_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:ES_rate_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:DSM_trick_mode_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:additional_copy_info_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_CRC_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_extension_flag"/>
<stx:template match="mp2:pack/mp2:PES_packet/
     mp2:video_stream/mp2:PES_header_data_length"/>
<stx:template match="mp2:pack/mp2:PES_packet/
```

```
       mp2:video_stream/mp2:PES_header_data_payload"/>
</stx:transform>
```

Listing D.8: A proof-of-concept STX stylesheet for adding shot information to a BSD.

```
<stx:transform
     xmlns:stx="http://stx.sourceforge.net/2002/ns"
     xmlns:sf="http://stx.sourceforge.net/2003/functions"
     xmlns:jvt="h264_avc"
     pass-through="all"
     output-method="xml"
     strip-space="no">
  <!-- Declaration of variables. -->
  <!-- A boolean indicating whether the current NAL unit is
       allowed to be copied (only for dropping). -->
  <stx:variable name="copy_nal_unit" select="false()"/>
  <!-- A variable for keeping track of the total number of
       NAL units in the bitstream. -->
  <stx:variable name="nalu_count" select="0"/>
  <!-- A variable for keeping track of the number of pictures
       in the bitstream. -->
  <stx:variable name="pic_count" select="0"/>
  <!-- A boolean indicating whether a new picture has been
       found. -->
  <stx:variable name="pic_found" select="false()"/>
  <!-- A variable that contains the number of slices per
       picture. -->
  <stx:variable name="max_slices_per_picture" select="5"/>
  <!-- A variable for keeping track of the number of slices
       parsed of the current picture. -->
  <stx:variable name="slice_count" select="0"/>
  <!-- Boolean indicating whether a picture has been found
       that marks the beginning of a shot. -->
  <stx:variable name="shot_pic_found" select="false()"/>
  <!-- A variable containing a list of picture numbers.
       Each number identifies an I slice coded picture
       that is located near the start of a shot (before
       or after the beginning of a shot). -->
  <stx:variable name="shots" select="(1,50,66,106,154,210,290,
       306,346,362,378,426,434,466,482,498,514,554,602,618,666,
       690,722,738,770,786,794,810,946,994,1034,1082,1114,1154,
       1186,1250,1290,1338,1386,1394,1426,1442,1458,1474,1490,
       1506,1546,1594,1610,1626,1634,1698,1714,1754,1770,1786,
       1802,1818,1834,1850,1890,1906,1922,1954,1970,2010,2058,
       2106,2114,2146,2178,2226,2234,2250,2266,2282,2298,2314,
       2330,2370,2418,2434,2474,2522,2570,2610,2626,2642,2690,
       2730,2746,2762,2778,2794,2834,2866,2882,2930,2946,2986,
     3034,3066,3106,3154,3186,3210,3242,3274,3290,3330,3378,
     3434,3498)"/>
<!-- Declaration of buffers. -->
<!-- A buffer for the storage of a complete NAL unit. -->
<stx:buffer name="nal_unit"/>
<!-- A buffer for the storage of the payload of a NAL unit;
     all transformations are done in this buffer. -->
<stx:buffer name="payload"/>
<!-- Put a complete NAL unit in a buffer; modify if needed;
     and write the result to the output stream. -->
<stx:template match="jvt:byte_stream_nal_unit">
  <stx:result-buffer name="nal_unit" clear="yes">
    <stx:assign name="nalu_count" select="$nalu_count + 1"/>
    <stx:message>
      NALU <stx:value-of select="$nalu_count"/> processed.
    </stx:message>
    <!-- Copy everything from the source stream into the
         buffer. -->
    <stx:copy>
      <!-- Determine whether other templates generate a match
           during the copy operation to the buffer. -->
      <stx:process-children/>
    </stx:copy>
  </stx:result-buffer>
  <stx:process-buffer name="nal_unit"
       group="add_picture_and_shot_info"/>
</stx:template>
<!-- Check the value of first_mb_in_slice syntax element;
     take appropriate action. -->
<stx:template match="jvt:first_mb_in_slice">
  <stx:copy>
    <stx:value-of select="."/>
  </stx:copy>
  <stx:if test=". = 0">
    <stx:assign name="pic_count" select="$pic_count + 1"/>
    <stx:message>
      Picture <stx:value-of select="$pic_count"/> processed.
    </stx:message>
    <stx:assign name="pic_found" select="true()"/>
  </stx:if>
  <stx:else>
    <stx:assign name="pic_found" select="false()"/>
  </stx:else>
</stx:template>
<!-- Group for processing byte_stream_nal_units that contain
     the first macroblocks of a picture. -->
<stx:group name="add_picture_and_shot_info">
  <stx:template match="jvt:byte_stream_nal_unit"
              public="no">
      <stx:element name="byte_stream_nal_unit"
                   namespace="h264_avc">
        <stx:attribute name="pic_cnt" select="$pic_count"/>
        <stx:if test="$pic_count = $shots">
          <stx:attribute name="shot" select="'true'"/>
          <stx:assign name="shot_pic_found" select="true()"/>
          <stx:assign name="slice_count"
                      select="$slice_count + 1"/>
          <stx:message>Slice
               <stx:value-of select="$slice_count"/>
               of shot picture
               <stx:value-of select="$pic_count"/>
               processed.
          </stx:message>
        </stx:if>
        <stx:else>
          <stx:attribute name="shot" select="'false'"/>
          <stx:assign name="shot_pic_found" select="false()"/>
          <stx:assign name="slice_count" select="0"/>
        </stx:else>
        <stx:process-children
             group="add_picture_and_shot_info"/>
      </stx:element>
    </stx:template>
  </stx:group>
  <!-- The empty group, which is used to copy something to the
       output stream. -->
  <stx:group name="empty"/>
</stx:transform>
```

Appendix E

A prospective view of DIA Amendment 2

E.1 Introduction

In this dissertation, several issues were identified pertaining to BSD-based content adaptation using MPEG-21 BSDL. Some of these problems emerged during the development of BS Schemata for a number of media formats and are due to a lack of expressive power of MPEG-21 BSDL (see Chapter 4). Other shortcomings turned up during the use of BSDL's format-agnostic BintoBSD process for translating the structure of coded video bitstreams into an XML description, and are performance related (see Chapter 5).

The aforementioned issues were reported to the Multimedia Description Schemes (MDS) sub-group of MPEG. A solution for improving the performance of the BintoBSD process, based on the use of context management attributes (see Appendix C), was proposed, as well as a solution for dealing with start code emulation prevention bytes, used by coding formats such as VC-1 and H.264/AVC (see Chapter 2).

Other interested parties contributed to MPEG-21 BSDL as well. As a result, a number of extensions to MPEG-21 BSDL are included in the second amendment to the MPEG-21 DIA standard [123]. This amendment also incorporates several enhancements to other DIA tools.

E.2 BSDL features in DIA Amendment 2

This section highlights several potential features of DIA Amendment 2 that are relevant in the context of this research. Note that this new set of extensions to DIA was still under development at the time of writing (summer of 2006). As such, this appendix only reflects the spirit of this amendment regarding the newly standardized BSDL features. For a complete and accurate overview, we would like to refer the interested reader to the standards document itself.

E.2.1 Extensions to BSDL-2

In the second amendment to MPEG-21 DIA, BSDL-2 is extended with the following tools.

- **Context management.** The five attributes, which are discussed in more detail in Appendix C, are incorporated in BSDL-2. Using these attributes, it is possible for the BintoBSD process to achieve a minimal memory consumption and a constant BSD generation speed on the one hand, while still allowing the use of the entire XPath 1.0 specification on the other hand.
- **Detection of start code emulation prevention bytes.** An attribute is added to BSDL-2 that specifies the removal of start code emulation prevention bytes from the bitstream processed, guaranteeing a correct analysis of the syntax elements parsed. In other words, the employment of this attribute guarantees the creation of a correct BSD for the bitstream processed. The use of this attribute is illustrated in Listing E.1 for the H.264/AVC video coding format: during BSD generation, every coding pattern in the input bitstream, taking the form of 0x000003 and starting at a byte-aligned position, is replaced by the 0x0000 byte string in order to guarantee a correct parsing behavior.

  Listing E.1: Use of `bs2:removeEmPrevByte`.

  ```
  <xsd:schema bs2:removeEmPrevByte="000003 0000">
  ```

- **User-defined XPath variables**. User-defined XPath variables allow to cache frequently used node sets (e.g., the parameter sets in H.264/AVC) as object arrays. They are introduced to simplify the notation of XPath expressions on the one hand and to speed up their evaluation on the other hand by reducing the number of predicates and the length of the location steps. For example, the use of XPath variables allows to significantly simplify the XPath expressions employed in the BS Schema for H.264/AVC. Indeed, the XPath expression provided in Listing 4.11 of Chapter 4 can be rewritten to the expression shown in Listing E.2.

Listing E.2: User-defined XPath variables.

```
$seq_parameter_set_rbsp[$pic_parameter_set_rbsp
[../jvt:pic_parameter_set_id + 1]/
jvt:seq_parameter_set_id + 1]/
jvt:log2_max_frame_num_minus4
```

- **ECMAScript.** The vendor-neutral ECMAScript standard is used to provide a platform-independent extension mechanism to BSDL. This extension mechanism allows adding user-defined datatypes (i.e., non-normative datatypes) and user-defined functions to BSDL. Also, the normative support for ECMAScript eliminates the need for the non-normative `bs0:implementation` attribute.

- **Support for syntax elements with a dependent bit length**. The length of a particular syntax element, expressed in terms of a number of bits, may be dependent on the value of another syntax element. For example, the length of the `jvt:slice_group_change_cycle` syntax element, which is optionally conveyed by the `slice_header()` syntax structure in an H.264/AVC bitstream, is dependent on the value of three other syntax elements (two of these syntax elements are part of an SPS, a third syntax element is part of a PPS).

  The support for syntax elements with a dependent bit length, together with the newly introduced support for a logarithmic function with base two, makes the use of the `SliceGroupChangeCycleType` class unnecessary. This class was previously discussed in Section 4.4.3. The computation of the bit length of the `jvt:slice_group_change_cycle` syntax element, using the newly defined BSDL-2 features, is outlined in more detail in Listing E.3 (`$PicSizeInMapUnits` and `$SliceGroupChangeRate` are placeholders for XPath expressions).

  Further, Listing E.4 contains a simplified notation, made possible by the new BSDL features, that can be used for the computation of the bit length of the `frame_num` syntax element. This notation is an alternative to the use of an `xsd:union`/`bs2:ifUnion` language construct for describing the value space of `frame_num` (see Listing 4.7).

Listing E.3: Computation of the bit length of `jvt:slice_group_change_cycle`.

```
<xsd:element name="slice_group_change_cycle">
```

```
  <xsd:simpleType>
    <xsd:restriction base="bs1:bitstreamSegment">
      <xsd:annotation>
        <xsd:appinfo>
          <bs2:bitsLength
            value="ceiling(bs2:log2($PicSizeInMapUnits
                   div ($SliceGroupChangeRate + 1))"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Listing E.4: Computation of the bit length of `jvt:frame_num`.

```
<xsd:element name="frame_num">
  <xsd:simpleType>
    <xsd:restriction base="bs1:bitstreamSegment">
      <xsd:annotation>
        <xsd:appinfo>
          <bs2:bitsLength
            value="$seq_parameter_set_rbsp/
                   jvt:log2_max_frame_num_minus4 + 4))"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Finally, in [19], it is discussed how BFlavor is extended with support for the detection of start code emulation prevention bytes. This paper also introduces a second important extension to BFlavor, offering a solution for the problem of keeping track of multiple parameter sets that are not immediately activated after retrieving them from an H.264/AVC bitstream.

E.2.2 Extensions to BSDL-1

In the second amendment to MPEG-21 DIA, BSDL-1 is extended with the following features.

- **Byte alignment.** Several datatypes are added to BSDL-1 that enable reading a value from the input bitstream and writing padding bits to the output bitstream until this bitstream is aligned on a one byte (using `bs1:align8)`, two byte (using `bs1:align16`), or four byte (using `bs1:align32`) boundary. More precisely, BintoBSD reads bits from the input bitstream until it is aligned on respectively a one byte, two byte, or four byte boundary and instantiates an element with the value read from the bitstream. When the bitstream is already correctly aligned, BintoBSD does not read any bits but instantiates an empty element.

BSDtoBin encodes the indicated lexical value on the number of bits required such that the output bitstream is aligned on a one byte, two byte, or four byte boundary. For an empty element, BSDtoBin uses the default or fixed value declared in the BS Schema, if relevant. Lastly, BSDtoBin does not write any bits if the bitstream is already correctly aligned.

Listing E.5 illustrates the use of the `bs1:align8` data type, which is a standardized solution for the use of the non-normative `CabacAlignmentOneBitType` class discussed in Section 4.4.3.

Listing E.5: Use of `bs1:align8`.

```
<xsd:element name="cabac_alignment_one_bit" minOccurs="0"
     maxOccurs="1" fixed="255" type="bs1:align8"/>
```

- **Emulation prevention bytes.** An attribute is added to BSDL-1 that instructs the BSDtoBin process to insert start code emulation prevention bytes at appropriate places in a bitstream. This prevents a BSDtoBin Parser from generating evil bitstreams (in this case, bitstreams with false start codes). The use of this attribute is illustrated in Listing E.6 for the H.264/AVC video coding format.

Listing E.6: Use of `bs1:insertEmPrevByte`.

```
<xsd:element name="bitstream">
  <xsd:complexType>
    <xsd:sequence>
      <!-- ... -->
    </xsd:sequence>
    <xsd:attribute ref="bs1:insertEmPrevByte" default=
    "000000 00000300 000001 00000301
     000002 00000302 000003 00000303"
  </xsd:complexType>
</xsd:element>
```

- **Exp-Golomb codes.** The set of built-in BSDL-1 datatypes is extended with normative support for Signed and Unsigned Exponentional Golomb codes, making the use of the `UnsignedExpGolomb` and `SignedExpGolomb` classes unnecessary.
- **ECMAScript.** Because ECMAScipt is used to extend the set of built-in BSDL-1 datatypes, the BSDtoBin process needs to have support for this scripting language for the purpose of binarization.
- **Enhanced** `bs1:byteRange` **and** `bs1:bitstreamSegment` **datatypes.** These built-in datatypes are extended with two new attributes: `addressMode` and `addressUnit`. The value of the `addressUnit` property specifies whether the length of a bitstream segment is given in terms of bits or bytes. According to the value of the `addressMode` property, the offset is relative to the start of the bitstream or to the start of the bitstream segment described by the parent element.

E.2.3 Profiles

In the second amendment to MPEG-21 DIA, ECMAScript is employed for the implementation of user-defined functions and datatypes, while XPath 2.0 is proposed for the implementation of XPath variables. These tools represent a significant implementation cost, which may not be acceptable for a use case such as on-the-fly content adaptation by a streaming server. Therefore, a number of profiles are defined for the BintoBSD and BSDtoBin Parsers in order to meet the constraints of different use cases. Moreover, by defining profiles that are restricting the use of XPath, the BintoBSD process can be simplified as well (regardless of the newly introduced extensions).

Profiles for BintoBSD

BSDL-2 uses XPath 1.0 for constraints specified in BSDL-2 language features such as `bs2:nOccurs`, `bs2:if`, `bs2:length`, and `bs2:ifUnion`. Consequently, a BintoBSD Parser needs to include an XPath 1.0 processor, which represents a significant implementation cost for constrained execution environments. However, a number of media formats may be parsed without the need for XPath expressions. Therefore, a first profile is defined for BintoBSD that excludes the use of XPath, while still allowing the use of variables, resulting in expressions such as `bs2:nOccurs="$number"`.

Furthermore, by excluding the use of location paths in XPath expressions, it is also possible to significantly reduce the complexity and memory footprint of an XPath processor. This approach prevents a BintoBSD processor from having to store a partially instantiated BSD.

In short, the following profiles are defined for BintoBSD:

- In the **Simple Profile**, the use of the BSDL-2 language features `bs2:nOccurs`, `bs2:if`, `bs2:ifUnion`, and `bs2:length` is restricted to the use of lexical values (i.e., constants) and variables (taking the form of `$var` expressions). A BintoBSD processor does not have to include an XPath processor, nor does it need to handle a partially instantiated BSD in memory during the parsing process.
- In the **Baseline Profile**, the use of location steps in XPath expressions is prohibited. A BintoBSD Parser only needs to implement a subset of an XPath processor and does not need to handle a partially instantiated BSD in memory during the parsing process.
- The **Main Profile** contains all tools available in the first version of BSDL-2, including the new extensions of Amendment 2, but excluding the support for XPath 2.0 and ECMAScript.
- The **Extended Profile** supports XPath 2.0 and the use of ECMAScript for the implementation of user-defined functions and datatypes. The latter tools are incorporated in the second amendment to DIA.

Profiles for BSDtoBin

XPath expressions are not used by the language features of BSDL-1. Hence, it is not necessary to define profiles for BSDtoBin that aim at a restricted use of XPath. However, ECMAScript may still be employed for the implementation of user-defined datatypes.

To summarize, the following profiles are defined for BSDtoBin:

- The **Main Profile** contains all tools that are available in the first version of BSDL-1, including the new extensions of Amendment 2, but excluding the support for ECMAScript.
- The **Extended Profile** supports the use of ECMAScript for the implementation of user-defined datatypes.

Signaling of profiles

To signal the required profile in a BS Schema and BSD, two new attributes are defined: one at the level of a BS Schema for BintoBSD and one at the level of a BSD for BSDtoBin. This is illustrated in Listing E.7.

Listing E.7: Profile signaling.

```
<!-- Profile signaling in a BS Schema. -->
<xsd:schema bs2:requiredProfile="simple">

<!-- Profile signaling in a BSD. -->
<bitstream bs1:requiredProfile="simple">
```

References

[1] Multimedia Description Schemes Sub-group. White paper on MPEG-21 Digital Item Adaptation. MPEG-document ISO/IEC JTC1/SC29/WG11 N8083, Moving Picture Experts Group (MPEG), Montreux, Switzerland, April 2006. Available on `http://www.chiariglione.org/mpeg/technologies/mp21-dia/index.htm`.

[2] ITU-T and ISO/IEC JTC 1. Generic coding of moving pictures and associated audio information – Part 2: Video. ITU-T Rec. H.262 - ISO/IEC 13818-2 (MPEG-2 Video), November 1994.

[3] ITU-T. Video coding for low bit rate communication. ITU-T Rec. H.263; version 1, Nov. 1995; version 2, Jan. 1998; version 3, Nov. 2000.

[4] ISO/IEC JTC 1. Information technology – Coding of audio-visual objects – Part 2: Visual. ISO/IEC 14496-2 (MPEG-4 Visual version 1), April 1999; Amd. 1 (ver. 2), Feb., 2000; Amd. 2, 2001; Amd. 3, 2001; Amd. 4 (Streaming Video Profile), 2001; Amd. 1 to 2nd ed. (Studio Profile), 2001; Amd. 2 to 2nd ed., 2003.

[5] Gary J. Sullivan and Thomas Wiegand. SVC Requirements Specified by VCEG (ITU-T SG16 Q.6). JVT-document JVT-N027, Hong Kong, China, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, January 2005. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[6] Ian Burnett, Rik Van de Walle, Keith Hill, Jan Bormans, and Fernando Pereira. MPEG-21: Goals and Achievements. *IEEE Multimedia*, 10:60–70, October-December 2003.

[7] Anthony Vetro, Charilaos Christopoulos, and Touradj Ebrahimi. Universal Multimedia Access. *IEEE Signal Processing Mag.*, 20(2):16–16, March 2003.

[8] ISO/IEC JTC 1. Information technology – Multimedia Framework (MPEG-21) – Part 7: Digital Item Adaptation. ISO/IEC 21000-7:2004, 2004.

[9] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (third edition). W3C Recommendation, W3C, February 2004.

[10] Fernando Pereira. Content and context: two worlds to bridge. In *Proceedings of the Fourth International Workshop on Content-Based Multimedia Indexing (CBMI 2005)*, Riga, Latvia, June 2005.

[11] Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Sam Lerouge, Koen De Wolf, and Rik Van de Walle. BFlavor: a harmonized approach to media resource adaptation, inspired by MPEG-21 BSDL and XFlavor. *EURASIP Signal Processing - Image Communication*, 21(10):862–889, November 2006.

[12] Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Koen De Wolf, and Rik Van de Walle. Using Bitstream Structure Descriptions for the Exploitation of Multi-layered Temporal Scalability in H.264/MPEG-4 AVC's Base Specification. *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2005*, 3767:641–652, October 2005.

[13] Wesley De Neve, Davy De Schrijver, Davy Van Deursen, Peter Lambert, and Rik Van de Walle. Real-Time BSD-driven Adaptation Along the Temporal Axis of H.264/AVC Bitstreams. *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, 4261:133–143, October 2006.

[14] Peter Lambert, Wesley De Neve, Philippe De Neve, Ingrid Moerman, Piet Demeester, and Rik Van de Walle. Rate-Distortion Performance of H.264/AVC Compared to State-of-the-Art Video Codecs. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(1):134–140, January 2006.

[15] Peter Lambert, Wesley De Neve, Yves Dhondt, and Rik Van de Walle. Flexible macroblock ordering in H.264/AVC. *Journal of Visual Communication & Image Representation*, 17:358–375, January 2006.

[16] Davy De Schrijver, Chris Poppe, Sam Lerouge, Wesley De Neve, and Rik Van de Walle. MPEG-21 Bitstream Syntax Descriptions for Scalable Video Codecs. *Multimedia Systems Journal*, 11(5):403–421, June 2006.

[17] Peter Lambert, Davy De Schrijver, Davy Van Deursen, Wesley De Neve, Yves Dhondt, and Rik Van de Walle. A Real-time Content Adaptation Framework for Exploiting ROI Scalability in H.264/AVC. In *Lecture Notes in Computer Science - Advanced Concepts for Intelligent Vision Systems - ACIVS 2006*, volume 4179, pages 442–453, September 2006.

[18] Stefaan Mys, Peter Lambert, Wesley De Neve, Piet Verhoeve, and Rik Van de Walle. SNR Scalability in H.264/AVC Using Data Partitioning. *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, 4261:333–343, October 2006.

[19] Davy Van Deursen, Davy De Schrijver, Wesley De Neve, and Rik Van de Walle. A Real-Time XML-based Adaptation System for Scalable Video Formats. *Lecture Notes in Computer Science - Advances in Multimedia Information Processing - PCM 2006*, 4261:343–353, October 2006.

[20] Sarah De Bruyne, Wesley De Neve, Koen De Wolf, Davy De Schrijver, Piet Verhoeve, and Rik Van de Walle. Temporal Video Segmentation on H.264/AVC Compressed Bitstreams. In *Accepted for publication in Proceedings of the 13th International MultiMedia Modeling Conference (MMM 2007)*, Singapore, January 2007.

[21] Davy De Schrijver, Wesley De Neve, Koen De Wolf, Robbie De Sutter, and Rik Van de Walle. An Optimized MPEG-21 BSDL Framework for the Adaptation of Scalable Bitstreams. *Accepted for publication in Journal of Visual Communication & Image Representation*.

[22] Wesley De Neve, Frederik De Keukelaere, Koen De Wolf, and Rik Van de Walle. Applying MPEG-21 BSDL to the JVT H.264/AVC Specification in MPEG-21 Session Mobility Scenarios. In *Proceedings of the 5th International Workshop on Image Analysis for Multimedia Interactive Services*, page 4 pp, Lisboa, Portugal, April 2004.

[23] Wesley De Neve, Peter Lambert, Sam Lerouge, and Rik Van de Walle. Assessment of the Compression Efficiency of the MPEG-4 AVC Specification. In *Proceedings of SPIE/Electronic Imaging 2004*, volume 5308, pages 1082–1093, San Jose, California, USA, January 2004.

[24] Wesley De Neve, Sam Lerouge, Peter Lambert, and Rik Van de Walle. A Performance Evaluation of MPEG-21 BSDL in the Context of H.264/AVC. In *Proceedings of SPIE Annual Meeting 2004: Signal and Image Processing and Sensors*, volume 5558, pages 555–566, Denver, Colorado, USA, August 2004.

[25] Wesley De Neve, Koen De Wolf, Davy De Schrijver, and Rik Van de Walle. Using MPEG-4 scene description for offering customizable and interactive multimedia presentations. In *Proceedings of the 6th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 4 on CD–rom, Montreux, Switzerland, April 2005.

[26] Wesley De Neve, Dieter Van Rijsselbergen, Charles Hollemeersch, Jan De Cock, Stijn Notebaert, and Rik Van de Walle. GPU-Assisted Decoding of Video Samples Represented in the YCoCg-R Color Space. In *Proceedings of the 13th ACM International Conference on Multimedia*, pages 447–450, Singapore, November 2005.

[27] Wesley De Neve, Davy De Schrijver, Dieter Van de Walle, Peter Lambert, and Rik Van de Walle. Description-Based Substitution Methods for Emulating Temporal Scalability in State-of-the-Art Video Coding Formats. In *Proceedings of the 7th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 83–86, Incheon, Korea, April 2006. Korea Information Science Society.

[28] Wesley De Neve, Davy De Schrijver, Davy Van Deursen, and Rik Van de Walle. XML-Driven Bitstream Extraction Along the Temporal Axis of SMPTE's Video Codec 1. In *Proceedings of the 7th International Workshop on Image*

Analysis for Multimedia Interactive Services, pages 233–236, Incheon, Korea, April 2006. Korea Information Science Society.

[29] Frederik De Keukelaere, Wesley De Neve, Peter Lambert, Boris Rogge, and Rik Van de Walle. MPEG-21 Digital Item Processing Architecture. In S. Furnell and P. Dowland, editors, *Proceedings of Euromedia 2003*, pages 5–9, Plymouth, April 2003. Eurosis.

[30] Robbie De Sutter, Sam Lerouge, Wesley De Neve, Peter Lambert, and Rik Van de Walle. Advanced Mobile Multimedia Applications: using MPEG-21 and time-dependent Metadata. In *Proceedings of SPIE/ITCom 2003*, volume 5241, pages 147–156, Orlando, USA, September 2003.

[31] Koen De Wolf, Robbie De Sutter, Wesley De Neve, and Rik Van de Walle. Comparison of prediction schemes with motion information reuse for low complexity spatial scalability. In *Proceedings of SPIE/Visual Communications and Image Processing (VCIP 2005)*, volume 5960, pages 1911–1920, Beijing, China, June 2005. SPIE.

[32] Davy De Schrijver, Wesley De Neve, Koen De Wolf, and Rik Van de Walle. Generating MPEG-21 BSDL Descriptions Using Context-Related Attributes. In *Proceedings of the 7th IEEE International Symposium on Multimedia*, pages 79–86, Irvine, USA, December 2005.

[33] Sarah De Bruyne, Koen De Wolf, Wesley De Neve, Piet Verhoeve, and Rik Van de Walle. Shot Boundary Detection Using Macroblock Prediction Type Information. In *Proceedings of the 7th International Workshop on Image Analysis for Multimedia Interactive Services*, pages 205–208, Incheon, Korea, April 2006. Korea Information Science Society.

[34] Davy Van Deursen, Wesley De Neve, Davy De Schrijver, and Rik Van de Walle. BFlavor: an Optimized XML-based framework for Multimedia Content Customization. In *Proceedings of the 25th Picture Coding Symposium*, pages 6 on CD–rom, Beijing, China, April 2006.

[35] Davy De Schrijver, Wesley De Neve, Koen De Wolf, Stijn Notebaert, and Rik Van de Walle. XML-Based Customization Along the Scalability Axes of H.264/AVC Scalable Video Coding. In *Proceedings of 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, pages 465–468, Island of Kos, Greece, May 2006.

[36] Koen De Wolf, Davy De Schrijver, Wesley De Neve, and Rik Van de Walle. Adaptive Residual Interpolation: A Tool For Efficient Spatial Scalability In Digital Video Coding. In H.R. Arabnia, editor, *Proceedings of the 2006 International Conference on Image Processing, Computer Vision & Pattern Recognition (IPCV'06)*, volume 1, pages 131–137, Las Vegas, Nevada, USA, June 2006.

[37] Peter Lambert, Wesley De Neve, Davy De Schrijver, Yves Dhondt, and Rik Van de Walle. Using Placeholder Slices and MPEG-21 BSDL for ROI Extraction in H.264/AVC FMO-encoded Bitstreams. In *Proceedings of International Conference on Signal Processing and Multimedia Applications (SIGMAP 2006)*, pages 9–16, Setúbal, Portugal, August 2006.

[38] Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Jan De Cock, and Rik Van de Walle. On an Evaluation of Transformation Languages in a Fully XML-driven Framework for Video Content Adaptation. In *Proceedings of the 2006 International Conference on Innovative Computing, Information and Control (ICICIC 2006)*, volume 3, pages 213–216, Beijing, China, August 2006.

[39] Dieter Van Rijsselbergen, Wesley De Neve, and Rik Van de Walle. GPU-driven Recombination and Transformation of YCoCg-R Video Samples. In *Proceedings of the Fourth IASTED International Conference on Circuits, Signals, and Systems (IASTED CSS 2006)*, pages 21–26, San Franciso, California, USA, November 2006.

[40] Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Sarah De Bruyne, and Rik Van de Walle. Exploitation of Interactive Region of Interest Scalability in Scalable Video Coding by Using an XML-driven Adaptation Framework. In *Proceedings of the 2nd International Conference on Automated Production of Cross Media Content for Multi-channel Distribution*, pages 223–231, Leeds, UK, December 2006.

[41] Jeroen Bekaert, Patrick Hochstenbach, Wesley De Neve, Herbert Van de Sompel, and Rik Van de Walle. Suggestions concerning MPEG-21 Digital Item Declaration. MPEG-document ISO/IEC JTC1/SC29/WG11 m9755, Moving Picture Experts Group (MPEG), Trondheim, Norway, July 2003.

[42] Frederik De Keukelaere, Wesley De Neve, Robbie De Sutter, and Rik Van de Walle. Suggestions concerning MPEG-21 Digital Item Method Operations and their implementation. MPEG-document ISO/IEC JTC1/SC29/WG11 m9754, Moving Picture Experts Group (MPEG), Trondheim, Norway, July 2003.

[43] Davy De Schrijver, Wesley De Neve, and Rik Van de Walle. Context-related attributes for MPEG-21 BSDL. MPEG-document ISO/IEC JTC1/SC29/WG11 m12217, Moving Picture Experts Group (MPEG), Poznan, Poland, July 2005.

[44] Davy De Schrijver, Wesley De Neve, Frederik De Keukelaere, and Rik Van de Walle. BNB comments on 21000-7 PDAM/2. MPEG-document ISO/IEC JTC1/SC29/WG11 m12824, Moving Picture Experts Group (MPEG), Bangkok, Thailand, January 2006.

[45] Davy De Schrijver, Wesley De Neve, Frederik De Keukelaere, and Rik Van de Walle. Proposal for additional extensions to MPEG-21 BSDL. MPEG-document ISO/IEC JTC1/SC29/WG11 m12825, Moving Picture Experts Group (MPEG), Bangkok, Thailand, January 2006.

[46] Wesley De Neve, Davy De Schrijver, Davy Van Deursen, Frederik De Keukelaere, and Rik Van de Walle. An MPEG-21 BS Schema for the first version of H.264/MPEG-4 AVC. MPEG-document ISO/IEC JTC1/SC29/WG11 m12823, Moving Picture Experts Group (MPEG), Bangkok, Thailand, January 2006.

[47] Wesley De Neve, Davy De Schrijver, Davy Van Deursen, Frederik De Keukelaere, and Rik Van de Walle. MPEG-21 BS Schemata for MPEG-$\{1, 2\}$ Video and Systems, MPEG-4 Visual, and H.264/MPEG-4 AVC. MPEG-document ISO/IEC JTC1/SC29/WG11 m13213, Moving Picture Experts Group (MPEG), Montreux, Switzerland, April 2006.

[48] Sylvain Devillers, Davy De Schrijver, Wesley De Neve, and Joe Thomas-Kerr. Report of CE on BSDL extensions. MPEG-document ISO/IEC JTC1/SC29/WG11 m13637, Moving Picture Experts Group (MPEG), Klagenfurt, Austria, July 2006.

[49] Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Saar De Zutter, and Rik Van de Walle. An MPEG-21 BS Schema for the scalable extension of H.264/MPEG-4 AVC version 6 (Joint Scalable Video Model 6). MPEG-document ISO/IEC JTC1/SC29/WG11 m13963, Moving Picture Experts Group (MPEG), Hangzhou, China, October 2006.

[50] ITU-T. Codec for videoconferencing using primary digital group transmission. ITU-T Rec. H.120; version 1, 1984; version 2, 1988; version 3, 1993.

[51] ITU-T. Video Codec for Audiovisual Services at px64 kbit/s. ITU-T Rec. H.261; version 1, Nov. 1990; version 2, Mar. 1993.

[52] ISO/IEC JTC 1. Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s – Part 2: Video. ISO/IEC 11172-2:1993, March 1993.

[53] Gary J. Sullivan and Thomas Wiegand. Video Compression - From Concepts to the H.264/AVC Standard. *Proc. the IEEE, Special Issue on Advances in Video Coding and Delivery*, 93(1):18–31, January 2005.

[54] Iain E. G. Richardson. *Video Codec Design: Developing Image and Video Compression Systems*. John Wiley & Sons, LTD, 2002.

[55] Joint Video Team (JVT) of ITU-T and ISO/IEC JTC 1. Advanced Video Coding: ITU-T Rec. H.264 and ISO/IEC 14496-10, version 1: May 2003, version 2: March 2004, version 3: July 2004, version 4: January 2005.

[56] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):560–576, July 2003.

[57] Special Issue on the H.264/AVC Video Coding Standard, *IEEE Trans. Circuits Syst. Video Technol.*, 13 (7) (2003).

[58] Special issue on the emerging H.264/AVC video coding standard, *Journal of Visual Communication & Image Representation*, 17 (2) (2006).

[59] Gary J. Sullivan. The H.264/MPEG-4 AVC video coding standard and its deployment status. In *Proceedings of SPIE/Visual Communications and Image Processing (VCIP 2005)*, volume 5960, pages 709–719, Beijing, July 2005. SPIE.

[60] Gary J. Sullivan, Pankaj Topiwala, and Ajay Luthra. The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. In *Proceedings of SPIE annual meeting 2004: Signal and Image Processing and Sensors*, volume 5558, pages 454–474, Denver, August 2004.

[61] Detlev Marpe, Steve Gordon, and Thomas Wiegand. H.264/MPEG4-AVC Fidelity Range Extensions: Tools, Profiles, Performance, and Application Areas. In *IEEE International Conference on Image Processing (ICIP'05)*, Genova, Italy, September 2005.

[62] Henrique Malvar and Gary J. Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. JVT-document JVT-I014, Trondheim, Norway, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, July 2003. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[63] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, LTD, 2003.

[64] Mathias Wien. *Variable Block-Size Transforms for Hybrid Video Coding*. Ph.D. dissertation, Rheinisch-Westfälischen Technischen Hochschule Aachen, 2004.

[65] Markus Flierl and Bernd Girod. Generalized B Pictures and the Draft H.264/AVC Video-Compression Standard. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):587–597, July 2003.

[66] Marta Karczewicz and Ragip Kurceren. The SP- and SI-frames design for H.264/AVC. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):637–644, July 2003.

[67] Stephan Wenger and Michael Horowitz. Flexible MB ordering - A new error resilience tool for IP-based video. In *International Workshop on Digital Communications (IWDC 2002)*, Capri, Italy, September 2002.

[68] Yves Dhondt, Peter Lambert, Stijn Notebaert, and Rik Van de Walle. Flexible macroblock ordering as a content adaptation tool in H.264/AVC. In *Proceedings of the SPIE/Optics East conference*, page 9 pp, Boston, October 2005.

[69] Solomon W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, 12(3):399–401, July 1966.

[70] Thomas Wiegand, Heiko Schwarz, Anthony Joch, Faouzi Kossentini, and Gary J. Sullivan. Rate-Constrained Coder Control and Comparison of Video Coding Standards. *IEEE Trans. Circuits Syst. Video Technol.*, 13(7):688–703, July 2003.

[71] Requirements for AVC Codec. JVT-document JVT-C156, Fairfax, Virginia, USA, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, May 2002. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[72] Siwei Ma, Wen Gao, Yan Lu, and Hanqing Lu. Proposed draft description of rate control on JVT standard. JVT-document JVT-F086, Awaji, Japan, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, December 2002. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[73] Peter Lambert. *FMO-Based Error Resilience and Adaptivity in H.264/AVC*. Ph.D. dissertation, Ghent University, 2007.

[74] Thomas Wedi and Yoshiichiro Kashiwagi. Subjective quality evaluation of H.264/AVC FRExt for HD movie content. JVT-document JVT-L033, Redmon, Washington, USA, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, July 2004. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[75] Thomas Stockhammer and Miska M. Hannuksela. H.264/AVC Video for Wireless Transmission. *IEEE Wireless Commun. Mag.*, 12(4):6–13, August 2005.

[76] Yves Dhondt, Stefaan Mys, Peter Lambert, and Rik Van de Walle. An evaluation of flexible macroblock ordering in error-prone environments. In Susanto Rahardja, JongWon Kim, Qi Tian, and Chang Wen Chen, editors, *Proceedings of the SPIE/Optics East Conference*, volume 6391, page 10 pp., Boston, 10 2006.

[77] Stephan Wenger and Michael Horowitz. FMO 101. JVT-document JVT-D063, Klagenfurt, Austria, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, July 2002. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[78] Stefaan Mys, Yves Dhondt, Dieter Van de Walle, Davy De Schrijver, and Rik Van de Walle. A performance evaluation of the data partitioning tool in h.264/avc. In Susanto Rahardja, JongWon Kim, Qi Tian, and Chang Wen Chen, editors, *Proceedings of the SPIE/Optics East Conference*, volume 6391, page 10 pp., Boston, 10 2006.

[79] Barry G. Haskell, Atul Puri, and Arun N. Netravali, editors. *Digital Video: An Introduction to MPEG-2*. Springer, December 1996.

[80] Dong Tian, Miska M. Hannuksela, and Moncef Gabbouj. Sub-sequence video coding for improved temporal scalability. In *Proceedings 2005 IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, pages 6074–6077, Kobe, Japan, May 2005.

[81] Miska M. Hannuksela. Enhanced concept of a GOP. JVT-document JVT-B042, Geneva, Switzerland, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, January 2002. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[82] Ville-Pekka Limnell, Dong Tian, Miska M. Hannuksela, and Moncef Gabbouj. Quality Scalability in H.264/AVC Video Coding. In *Proceedings of SPIE/Visual Communications and Image Processing (VCIP 2005)*, pages 559–567, Beijing, China, July 2005.

[83] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Hierarchical B pictures. JVT-document JVT-P014, Poznan, Poland, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, July 2005. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[84] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Analysis of Hierarchical B Pictures and MCTF. In *Proceedings of the IEEE International Conference on Multimedia & Expo (ICME 2006)*, Toronto, Canada, July 2006.

[85] Jens-Rainer Ohm. Advances in scalable video coding. *Proc. IEEE*, 93(1):42–56, January 2005.

[86] Julien Reichel, Heiko Schwarz, and Mathias Wien. Scalable Video Coding - Joint Draft 4. JVT-document JVT-Q201, Nice, France, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, October 2005. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[87] Julien Reichel, Heiko Schwarz, and Mathias Wien. Joint Scalable Video Model JSVM-4. JVT-document JVT-Q202, Nice, France, Joint Video Team of ISO/IEC JTC1/SC29/WG11 and ITU-T SG16/Q.6, October 2005. Available on `http://ftp3.itu.int/av-arch/jvt-site`.

[88] ISO/IEC JTC 1. Information technology – JPEG 2000 image coding system: Core coding system.

[89] ISO/IEC JTC 1. Information technology – Coding of audio-visual objects – Part 3: Audio.

[90] Debargha Mukherjee, Eric Delfosse, Jae-Gon Kim, and Yong Wang. Optimal Adaptation Decision-Taking for Terminal and Network Quality-of-Service. *IEEE Trans. Multimedia*, 7(3):454–462, June 2005.

[91] João Magalhães and Fernando Pereira. Using MPEG standards for multimedia customization. *Signal Processing: Image Communication*, 19(5):437–456, May 2004.

[92] Sam Lerouge, Peter Lambert, and Rik Van de Walle. Multi-criteria Optimization for Scalable Bitstreams. In *Proceedings of the 8th International Workshop on Visual Content Processing and Representation*, pages 122–130, Madrid, Spain, September 2003. Springer.

[93] Gabriel Panis, Andreas Hutter, Joerg Heuer, Hermann Hellwagner, Harald Kosch, Christian Timmerer, Sylvain Devillers, and Myriam Amielh. Bitstream syntax description: a tool for multimedia resource adaptation within MPEG-21. *Signal Processing: Image Communication*, 18(8):721–747, September 2003.

[94] SMPTE. Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process. SMPTE 421M-2006, 2006.

[95] ISO/IEC JTC 1. Information technology – Generic coding of moving pictures and associated audio information – Part 1: Systems. ISO/IEC 13818-1:2000, 2000.

[96] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer (second edition). W3C Recommendation, W3C, October 2004.

[97] James Clark and Steve DeRose. XML Path Language (Version 1.0). W3C Recommendation, W3C, November 1999.

[98] James Clark. XSL Transformations (XSLT) version 1.0. W3C Recommendation, W3C, November 1999.

[99] B.S. Manjunath, Philippe Salembier, and Thomas Sikora. *Introduction to MPEG-7: Multimedia Content Description Interface*. Wiley, New Jersey, 2003.

[100] Sam Lerouge. *Personalizing Quality Aspects for Video Communication in Constrained, Heterogeneous Environments*. Ph.D. dissertation, Ghent University, 2005. Available on `http://www.firw.ugent.be/doctoraat/doctoraten/documenten/tekst/Lerouge_PhD.pdf`.

[101] Kal Ahmed, Danny Ayers, Mark Birbeck, Jay Cousins, David Dodds, Josh Lubell, Miloslav Nic, Daniel Rivers-Moore, Andrew Watt, Robert Worden, and Ann Wrightson. *XML Meta Data*. Wrox, 2001.

[102] Joseph Thomas-Kerr, Ian Burnett, and Christian Ritz. Format-independent Multimedia Streaming. In *Proceedings of IEEE International Conference on Multimedia & Expo (ICME 2006)*, Toronto, Canada, July 2006.

[103] Danny Hong and Alexandros Eleftheriadis. XFlavor: Bridging Bits and Objects in Media Representation. In *Proceedings of the IEEE International Conference on Multimedia & Expo (ICME 2002)*, Lausanne, Switzerland, August 2002. Available on `http://flavor.sourceforge.net/`.

[104] Xiaoming Sun, Chang-Su Kim, and C.-C. Jay Kuo. MPEG video markup language and its applications to robust video transmission. *Journal of Visual Communication and Image Representation*, 16(4-5):589–620, August-October 2005.

[105] Ian Burnett, Fernando Pereira, Rik Van de Walle, and Rob Koenen, editors. *The MPEG-21 Book*. Wiley, March 2006.

[106] Special Issue on MPEG-21, IEEE Trans. Multimedia 7 (3) (2005).

[107] Myriam Amielh and Sylvain Devillers. Bitstream Syntax Description Language: Application of XML-Schema to Multimedia Content Adaptation. In *WWW2002: The Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002. Available on `http://www2002.org/CDROM/alternate/334/`.

[108] Gauthier Lafruit, Eric Delfosse, Roberto Osorio, Wolfgang van Raemdonck, Vissarion Ferentinos, and Jan Bormans. View-Dependent, Scalable Texture Streaming in 3-D QoS With MPEG-4 Visual Texture Coding. *IEEE Trans. Circuits Syst. Video Technol.*, 14(7):1021–1031, July 2004.

[109] Sylvain Devillers, Christian Timmerer, Joerg Heuer, and Hermann Hellwagner. Bitstream Syntax Description-Based Adaptation in Streaming and Constrained Environments. *IEEE Trans. Multimedia*, 7(3):463–470, June 2005.

[110] Sylvain Devillers and Eric Delfosse. Non-normative features for BSDL and update on reference software. MPEG-document ISO/IEC JTC1/SC29/WG11 M9804, Moving Picture Experts Group (MPEG), Trondheim, Norway, July 2003.

[111] Alexandros Eleftheriadis. Flavor: A Language for Media Representation. In *ACM Multimedia Conf.*, pages 1–9, Seattle, WA, November 1997. Available on `http://flavor.sourceforge.net/`.

[112] Yu Lu, Feng Yi, Jie Dong, and Cixun Zhang. Overview of AVS-video: tools, performance and complexity. In *Proceedings of SPIE/Visual Communications and Image Processing (VCIP 2005)*, volume 5960, pages 679–690, Beijing, China, June 2005. SPIE.

[113] Sridhar Srinivasan, Pohsiang (John) Hsu, Tom Holcomb, Kunal Mukerjee, Shankar L. Regunathan, Bruce Lin, Jie Liang, Ming-Chieh Lee, and Jordi Ribas-Corbera. Windows Media Video 9: overview and applications. *Signal Processing: Image Communication*, 19(9):851–875, October 2004.

[114] Youngsun Lee, Jinwhan Lee, Hyunsik Chang, and Jae Yeal Nam. A New Scene Change Control Scheme based on Pseudo-skipped Picture. In *Proceedings of SPIE/Visual Communications and Image Processing*, volume 3024, pages 159–166, San Jose, CA, USA, January 1997. SPIE.

[115] ISO/IEC JTC 1. Information technology – MPEG systems technologies – Part 1: Binary MPEG format for XML. ISO/IEC 23001-1:2006, 2006.

[116] Gerrard Drury and Joseph Thomas-Kerr. ISO/IEC 21000-18 CD MPEG-21 Digital Item Streaming. MPEG-document ISO/IEC JTC1/SC29/WG11 N7739, Moving Picture Experts Group (MPEG), Nice, France, October 2005. Available on `http://www.chiariglione.org/mpeg/working_documents.htm`.

[117] Petr Cimprich et al. Streaming Transformations for XML (STX). Technical report, Available on `http://stx.sourceforge.net/documents/spec-stx-20040701.html`, 2004.

[118] Shih-Fu Chang and Anthony Vetro. Video adaptation: Concepts, technology, and open issues. *Proc. IEEE*, 93(1):145–158, January 2005.

[119] Makoto Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *Proceedings of the Twelfth International Conference on*

Information and Knowledge Management, pages 342–349, New Orleans, LA, USA, 2003. ACM Press.

[120] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of the 19th International Conference on Data Engineering*, pages 455–467, Bangalore, India, 2003.

[121] Toni Zgaljic, Nikola Sprljan, and Ebroul Izquierdo. Bitstream Syntax Description Based Adaptation of Scalable Video. In *Proceedings of the European Workshop on the Integration of Knowledge, Semantic and Digital Media Technologies (EWIMT)*, London, UK, 2005.

[122] Peter Lambert, Dieter Van de Walle, Wesley De Neve, and Rik Van de Walle. ROI Scalability in H.264/AVC's Base Specification. In *Submitted to Visual Communications and Image Processing 2007 (VCIP 2007)*, San Jose, California, USA, February 2007.

[123] ISO/IEC JTC 1. Dynamic and distributed adaptation. ISO/IEC 21000-7:2004/Amd.2 (work in progress), 2006.