# EE573 project report

# OO analysis/design of CABAC, and UML modeling of H.264 encoder

Hassan Shojania
shojania@ieee.org

December 17, 2004

# 1. Introduction

## H.264 background/history:

H.264, also known as Advanced Video Coding (AVC), is the newest video coding standard developed jointly by VCEG (Video Coding Experts Group) of ITU (International Telecommunication Union) and MPEG (Moving Pictures Experts Group) of ISO (International Standard Organization) under JVT (Joint Video Team) name and released in 2003. This standard has also been released under part 10 of MPEG-4 to complement MPEG-4 Visual (part 2 of MPEG-4) which was released in 2000. MPEG-4 standard is an ambitious standard which goes beyond delivering traditional sequences of 2D video & audio and constructs a scene as hierarchy of elements from video, audio, text, synthetic objects, … where every element is treated as an object. Not only this provides better flexibility in choice of compression techniques, it makes features like interactive media, animation, personalized enhancements to a video program, … possible. Though both MPEG-4 Visual (part 2 of MPEG-4) and H.264/AVC are concerned with compression of visual data, H.264/AVC is focused on high efficiency while MPEG-4 Visual on flexibility. One of the reasons that MPEG-4 hasn't become as much success as MPEG-2 (established in 1994) is its limited improvement of compression efficiency for broadcast applications. H.264/AVC was meant to address this deficiency.

These standards are not written as development guides but with the goal of ensuring compatibility and interoperability between different implementations. For example an AVC/H.264 compliant bitstream generated by one encoder must be decodable by another manufacturer's decoder (i.e. high coupling). The standard describes the decoding process and the bitstream format in details while encoding process is not specified at all in order to allow designers to choose their own method of encoding. These are complex documents; e.g. MPEG-4 Visual consists of 539 pages and H.264/AVC is over 250 pages. Since the standards are difficult to interpret, reference software is also developed by the standard bodies (though not optimized, and meant as a proof of concept) to facilitate implementation of the standard by different people/companies and ensure same understanding of standard specification.

H.264/AVC is employing many new techniques to achieve higher compression ratio. One is a new entropy coding technique called CABAC (Context Adaptive Binary Arithmetic Coding) performing the last stage of encoding (virtually) before bitstream generation. Because of high utilization of this stage (an average of 50 million bits per second with a possible maximum instantaneous rate of 800 million bits per second), CABAC is an important part of H.264/AVC in any hardware/software implementations.

Our focus in this project is on H.264/AVC reference software. It's a huge software (over 65,000 lines of code) written in C, without any documentation

and very minimal in-line comments. It is updated regularly since release of the standard to cover more parts of the standard and also fix existing bugs. The software consists of an encoder (to generate a conforming bitstream according to the standard) and a matching decoder (conforming to the decoding process described in the standard); it is capable of encoding an input video sequence (in YUV 4:2:0 color format) and decoding the encoded sequence back to the original sequence.

## Why this project?

The author's thesis is about implementation of a high-speed hardware of CABAC with proper programming model and software interface to the rest of H.264 codec. Since start of this thesis, there have been many struggles with the reference software. Though the code is modularized by grouping related functions in separate files and grouping related data in data structures, it still very much lacks the characteristics and clarity of a good OO design. As there is no information hiding and use of global variables is common, side-effects of function calls are not known. Though the focus has been mainly on CABAC, changes made to CABAC portion (e.g. inserting a HW-model code in place of original code) resulted in many issues (e.g. because of different sequence of usage, synchronization, …) solely because of not knowing the details of other portion of the software. There had been many hours spent on debugging huge scary code to resolve simplest issues. Even after passing this long learning curve, it was realized that the hardly gained knowledge was dissipating with a high rate! Even only a week or two was enough to forget many details because what learned previously was not documented. On the other hand, a few attempts for gradual textual documentation of whatever learned failed as priority was to solve the main issues not documenting code that wasn't directly related to the goal. What was really needed was a long commitment to solely spend on modeling and documentation of the rest of codec. Of course it wasn't justifiable much to take a minimum 3 weeks off from the thesis and spend on this (though in a long run would worth it). This course project was a great opportunity to justify this time and commitment!

In this project, we intend to analyze CABAC, model and design an OO CABAC using UML. We also intend to provide a high level UML model of H.264/AVC encoder (though not complete). As the reference software is not object oriented and without any documentation, we believe this effort can help with better understanding of the code and eventually the standard itself for future references by the author and other students. At the same time, it serves as a good exercise of what we have learned so far about OO analysis and design along UML concepts.

# 2. Video compression background

Here we briefly go through some terms and basics ideas in video compression to allow better understanding of the later topics. This section can be skipped at this point and only referred to when needed.

**Codec:** It is an acronym for en**co**der/**dec**oder pair.

**Asymmetric coding:** As usually a video content is compressed once but decompressed (played back) several times by a larger group of people, the decompression process should be as light as possible. Popular video compression algorithms (MPEG 1/2/4), all require way more computation power on the encoding side compared to the decoding side (in order of tens of times). Naturally some degree of HW acceleration for encoding process always needed.

**Video compression:**

Video compression is achieved through different sources. Below describes some of the main ways to achieve compression:

**Intra-transformation:** Within a frame, there exists spatial redundancy in color of pixels in any neighborhood (depending on the region of frame). This correlation is exploited to achieve compression by some mathematical transformations (e.g. Discrete Cosine Transform). A block of pixels is transformed to an equivalent presentation of the same block but with generating high deviation between value of colors at different location of the block mainly generating high values at upper-left corner of the block and many zero or close to zero values at bottom-right area. By dropping less-significant values, the amount of data is reduced but without much decrease of quality. A similar but inverse process called inverse transformation reconstructs the block at the decompression time.

**Inter-prediction:** There is lots of temporal redundancy between successive frames of a content as fast movement of camera or objects within frame doesn't happen very often (considering usual 30 frames per second video sequence). By detecting movement of blocks of pixels from one frame to another, instead of transmitting the whole block just a motion vector representing the motion is sent (of course this is a simplified version of the process as more complexity is involved).

**Entropy-coding:** This stage is almost (except NAL stage) before generation of the final bitstream. This coding technique exploits redundancy between codes generated by higher level encoders. By assigning a shorter code to a more frequent element and longer to less frequent one, the average length of generated stream is reduced. By trying to capture statistical characteristics of elements and adapting this code assignment, better compression ratio can be achieved (*context adaptive*).

Below figure shows major units of H.264 encoder and data flow between them. $F_n$ is the n-th frame, $F'_{n-1}$ is the reconstructed reference frame used for motion estimation, and $F'_n$ is the reconstruction of current n-th frame. ME (Motion Estimation) and MC (Motion Compensation) relates to inter-prediction already

described. T and $T^{-1}$ are the forward and inverse transformations. Q and $Q^{-1}$ are quantization and de-quantization blocks.

Depending on the type of slice current frame (Fn) is encoded based on it (I/B/P/SI/SP), either inter or intra prediction is selected. The prediction macro/subblock is subtracted from the original macro/subblock to form the residual. This residual (which has much smaller dynamic range) is transformed and quantized. The resulted coefficients are reordered and entropy coded and finally packed for network transmission. Same macro/subblock is also reconstructed, so its reconstructed value is used for neighboring blocks (intra) or as reference for intra prediction.

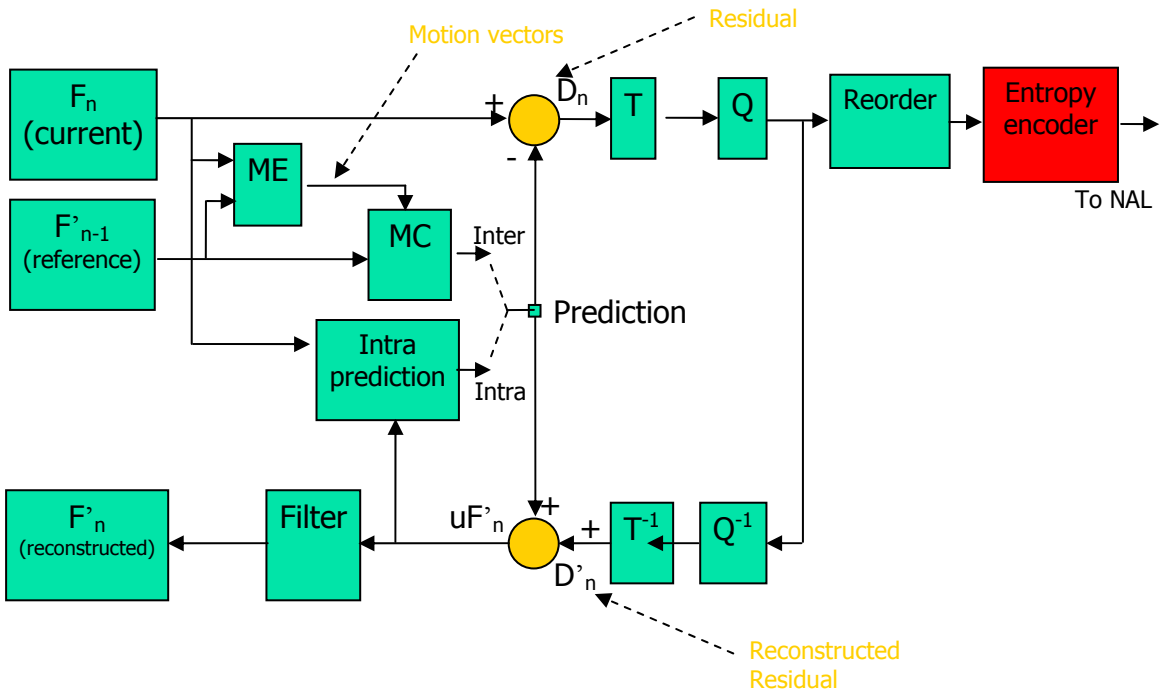


Figure 1. H.264 encoder building blocks along data-flow between them

**CABAC versus CAVLC:**

H.264 targets different applications with different requirements so it supports different *profiles*: *Baseline* profile targets videotelephony, videoconferencing and wireless applications. *Main* profile targets television broadcasting and video storage. *Extended* profile targets streaming media applications.

Naturally the main profile requires the highest resolution, quality and bandwidth so needs maximum compression possible. This profile employs CABAC (beside other features) at entropy coding level which beats other profiles' CAVLC coder (similar to traditional entropy coders used in earlier standards) by 9-14%. This coder is way more computationally expensive than CAVLC especially at the high resolution/bitrate main profile is targeted for. Its level 2.1 requires coding of 50 million bits per second which could go up to instantaneous bitrate of 800

millions per second. Since CABAC employs binary arithmetic coding at its heart (which gives its higher compression ratio), its coding is highly serial and can not be parallelized. That is why a hardware implementation of whole or partial hardware acceleration is required for it.

# 3. Analysis of H.264 codec

The requirements for a CABAC decoder is specified in section 9.3 of H.264 Standard Specifications [ITU`03]. This is similar to the rest of the standard where only decoding process of a compliant stream is described and encoding process needs to be inferred from the decoding process. For CABAC, some flowcharts of encoding stages are given in the standard only as recommendation though. In short, a CABAC encoder needs to generate a compliant bitstream that can be decoded using the CABAC decoder described in section 9.3 (along all the tables and flowchart of decoding process). For the reference software, the standard specification becomes the major SRS (Software Requirement and Specification) too as the whole standard is implemented in software (i.e. no hardware acceleration). Since the reference SW is a proof of concept only, it doesn't intend to be fast, efficient or target a particular range of applications like a commercial product. It supports all items of the standard but some cases might take very long to execute (depending on the speed of system). It also doesn't have any UI, video source is a file, similarly a file as output of codec. No particular platform is targeted, so no platform dependent features are used; only plain C functions.

Fortunately the existence of the reference software beside the standard was very helpful to understand the underlying specification and algorithms. The author was using the reference software as whole and particularly the CABAC-related portion for roughly a month before start of this project. The CABAC portion was suffering from similar issue as rest of the software described in the introduction portion of this report, mainly all drawbacks of a large software written in a procedural language which also lacks documentation. Understanding the behavior, data flow and call graph of different modules required tens of hours of debugging while referring to standard and books. When change of some portion of CABAC code started (to plug in some experimental hardware model and test the validity of the model), whole new issues started to show up as some call sequences were changed and side-effects of changes were unknown (e.g. because of lack of information hiding). At points, breakpoints on data access to a memory location was used to figure out where/when a member of a data structure was changing as there was no clue how its value was changing from one portion of code to another (also single stepping a code that running it as a whole was taking more than 3 hours was not an option). All these issues were suggesting of the usefulness of a model for whole reference software.

As an OO design and implementation of the whole reference software is within scope of a thesis itself, this project was limited to OO design of CABAC in

particular along UML model of more important portions of current reference software but with an OO emphasize.

Now to better represent where (what layer) CABAC sits in the reference codec, we go through some high level use cases of a media application down to the CABAC. Note that these use cases are not CABAC use cases and just shown to better illustrate the concept of video coding and familiarize the reader with the field.

### Use-case 1: Media application use-case

Below use case diagram shows some of the features of a media application and how a user will interact with it. Of course this doesn't cover all features and use cases of nowadays modern media application but it shows interaction with some other actors in fulfilling its functionalities.

The diagram itself is self-explaining enough so we don't repeat it as a use case specification as it's not our main focus here.

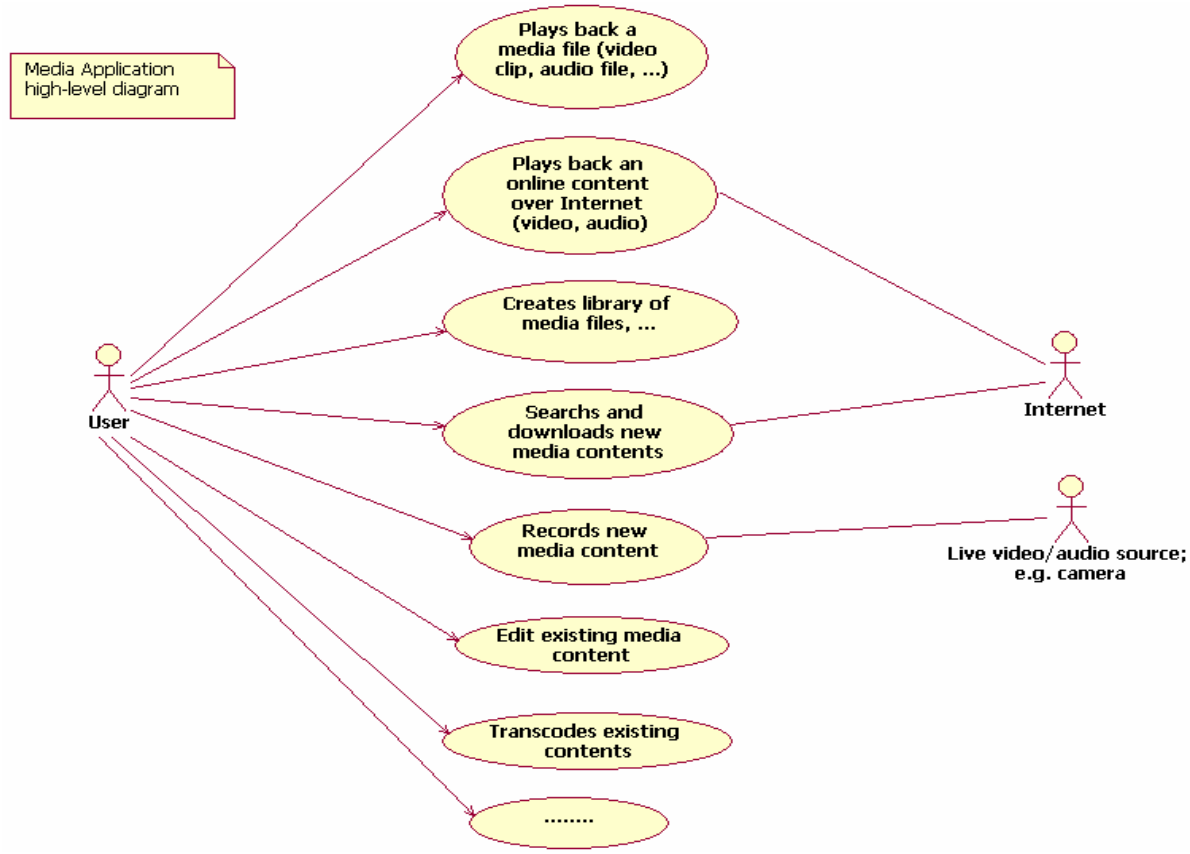


Figure 2. Media application use-case

## Use-case 2: Media application encoding/decoding use-case

This use case diagram focuses on encoding and decoding processes within a media application and its interaction with other actors. Again, here we refrain from presenting a formal use-case spec.

In the encoding process, first there is a source of video content (uncompressed) intended to be transformed to a compressed form. Note that encoding is different than recording (though might involve recording too) in the sense that it is not just capture of video but requires further processing (i.e. compression) too. Any compression involves tradeoff between quality (resolution, picture quality, jerkiness,…), encoding speed and storage/bandwidth size (depending if the output is stored or broadcasted). This requires adjustment of a set of "encoding parameters" which can be done either by the user in an interactive fashion (e.g. through a multi-step wizard), automatically by media application or through some predefined set of configurations. Of course, factors like a live content, availability of hardware support, efficiency of codec, … would limit the range of possible choices.

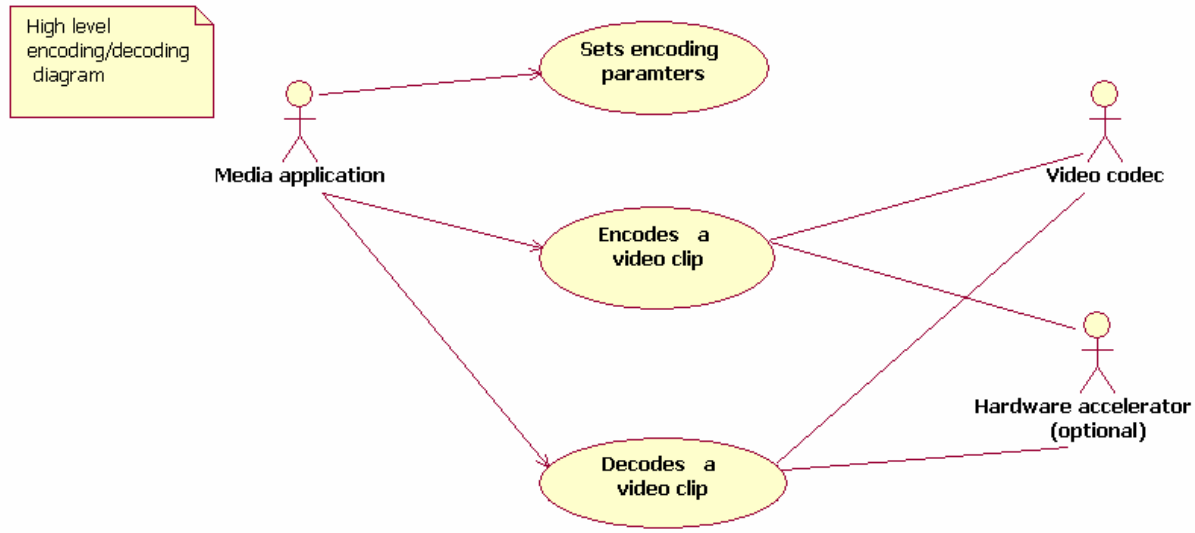


Figure 3. High-level encoding/decoding use-case

"Encode of a video content" involves other actors too. Here video codec is shown as an actor in a sense that nowadays a video codec as a whole is often a separate component deployed separately on a system than the media applications and if properly registered, it can support different media applications (e.g. RealPlayer, QuickTime, WindowsMedia, …). It supports interfaces for encoding and decoding but because of performance issue, it will be mainly implemented as an in-process library loaded by the media application. "Hardware accelerator" is shown as a potential actor here to emphasize the role hardware might play in encoding/decoding. Usually most of the hardware access is abstracted by the video codec itself rather than media application, but there

might be still some query of capabilities and degree of hardware support done by the media application. Media application receives the compressed content from the codec (gradually) and could use it in different scenarios (e.g. broadcasting, storage, streaming, …).

Similarly, decoding of a video content uses the decoding interface of the codec to decompress the compressed source and provide the media application with the uncompressed source. Then the media application will decide how to use this decompressed content (e.g. displaying, storage, …) based on higher level use-case of application at that point.

### Use-case 3: Media application encoding use-case (I/O view)

This use case diagram focuses mainly on I/O and interaction with other players during encoding process. Source of uncompressed data is either disk or live content. Reading/capturing this source is on going process and chunk of this content (e.g. in frame units) is passed to the encoding block which itself take advantage of codec and HW accelerator. At the last stage, the compressed data is prepared (e.g. packetized) for proper medium for streaming or storage.

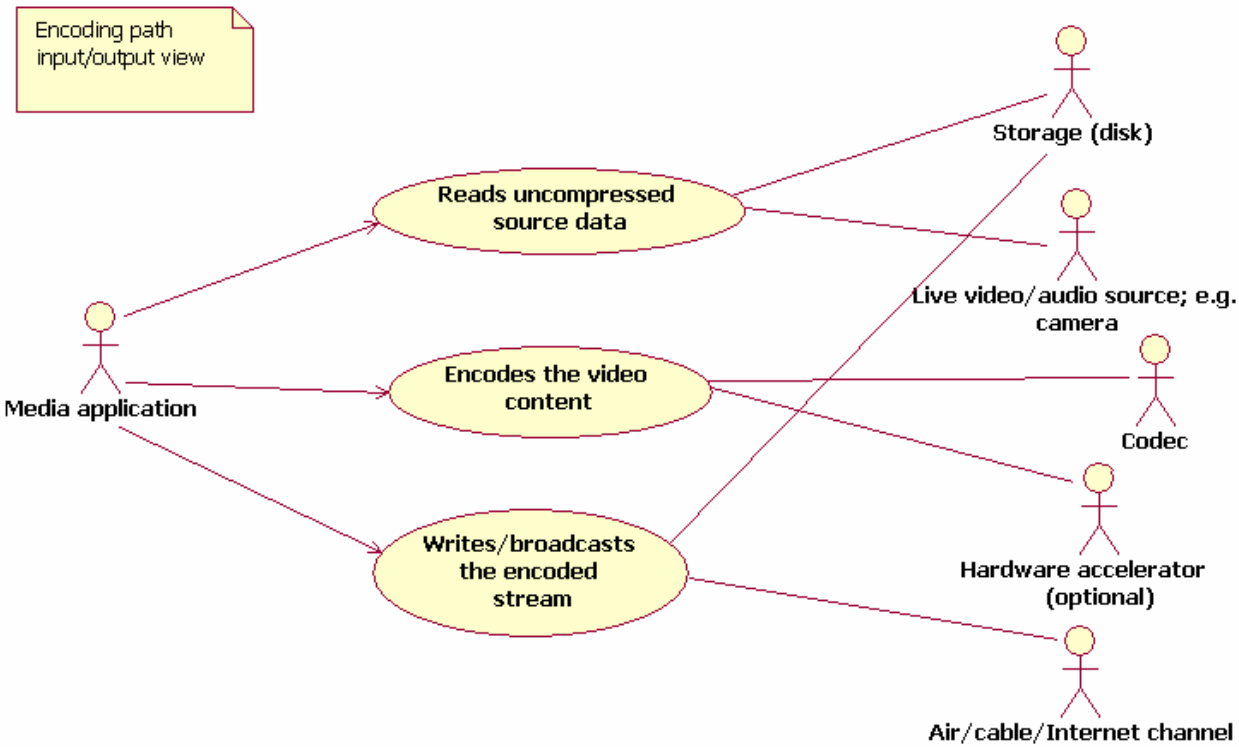


Figure 4. Encoding path input/output view

### Use-case 4: H.264 encoder use-case

This is a use case diagram depicting similar block diagram of Fig. 1. Comparing it against that block diagram is interesting; instead of focusing on data-flow of that figure, it shows equivalent functionalities. Virtually every operation in the reconstruction path (except the de-blocking filter) is the inverse of an operation in

the forward path. The encoder reconstructs every frame of the encoded video (similar to what the decoder does at playback time). This is done so intra prediction of neighboring macroblocks or inter-prediction of macroblocks in other frames will use the same data the decoder sees not the original data that's visible to encoder only (because of lossy nature of encoding).

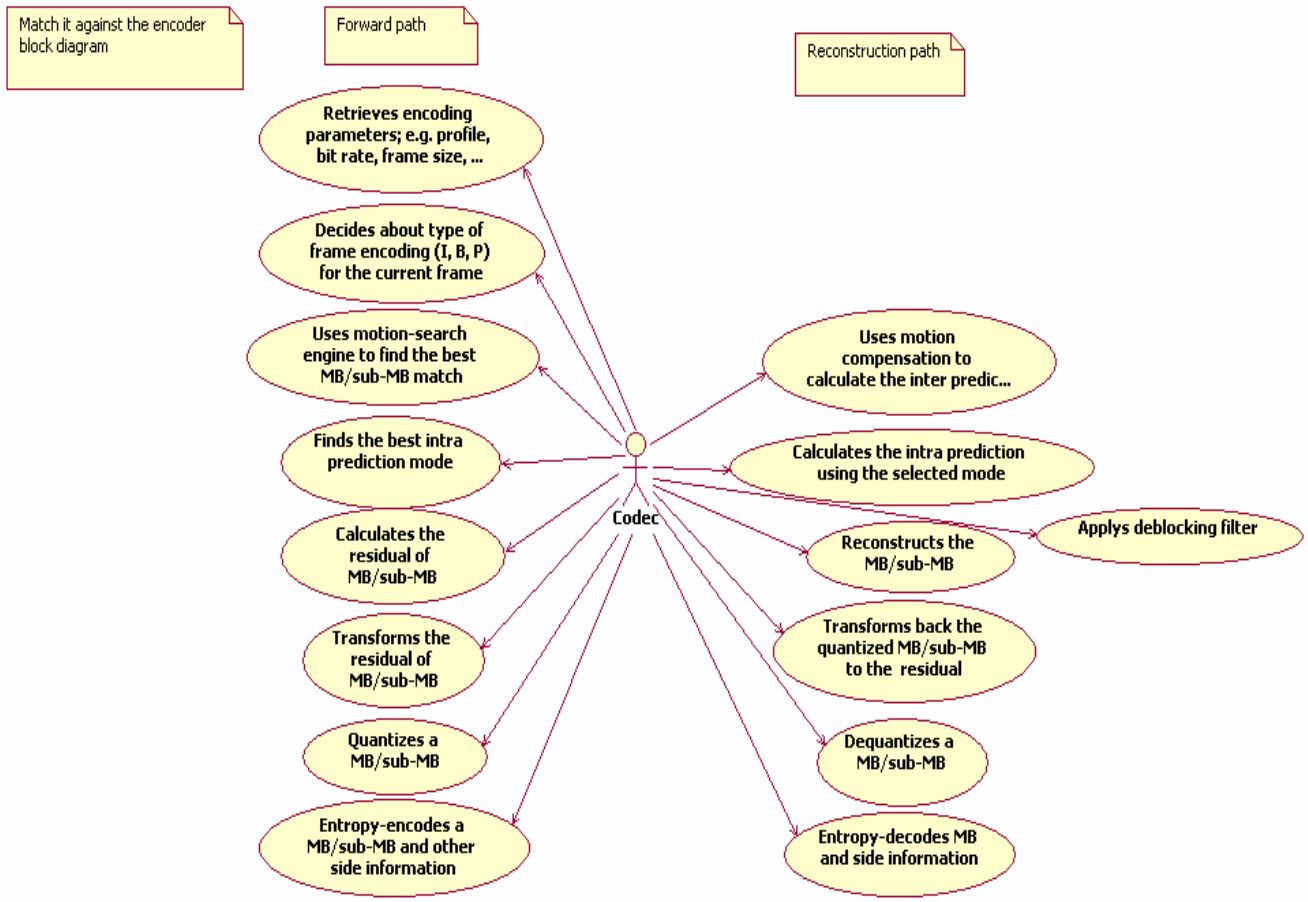


Figure 5. H.264 encoder use-case

### Use-case 5: Slice generator use-case

Each frame is encoded as one or more slices, a sort of highest level of encoding unit. Each slice is composed of many syntax elements (e.g. transform coefficient residuals, motion vector differences, macroblock type/subtype, header info, …). This syntax elements are generated after all higher level coding is done and they're ready to be entropy-coded. This higher level coding is done through either inter or intra prediction. Inter-prediction involves motion estimation which means comparing macroblocks of one slice against another's to find the closest match. This process requires lots of code which is more of low-level mathematical nature (also normally the main bottleneck of every encoder), thus would make sense to put it into a separate package. Macroblock is the biggest unit of prediction (for both intra/inter)

and involves a good portion of reference SW so would make sense to have it as a separate package. Also, rate-distortion is higher level decision making that chooses the best prediction (the one resulting in lowest bitrate considering trade-off between speed of decision making and quality/bitrate of compressed content), so it make sense to separate it into another package.

Either or both CAVLC and CABAC support is needed in a H.264 encoder depending on the profiles that particular encoder supports (*baseline* and *extended* profile require CAVLC, *main* profile requires CABAC). Though they're normally implemented as an integral part of main encoder (and not deployed as a separate component), we consider them as components here to emphasize the importance of a good interface between them and the rest of encoder. Otherwise they could have been packages instead.

After entropy coding, the resulted data called VCL (Video Coding Layer) data is mapped to NAL (Network Abstraction Layer) units prior to transmission or storage. The purpose of this layer is to distinguish between coding related features at VCL and transport related features at NAL.

Of course there is another higher level code that employs slice generator many times during the encoding process but that layer is pretty much thin and involves I/O (e.g. file read/write) and administration work (e.g. reading encoding configuration parameters).

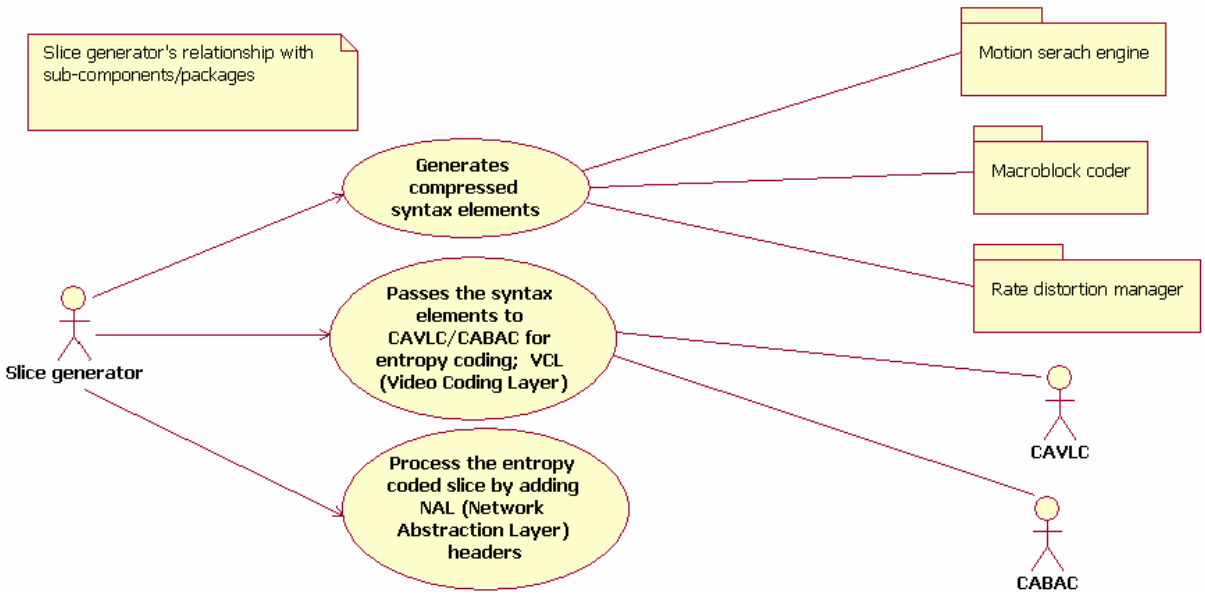


Figure 6. Slice generator's use-case

## CABAC

CABAC stands for Context Adaptive Binary Arithmetic Coding and consists of few sections itself:

**1) Binarizer** is a form of pre-processing stage (before coding) that operates on syntax elements and generates a unique *intermediate* binary codeword for a given syntax element. This intermediary codeword is called *bin string* and each binary value of it called a *bin*. This stage effectively reduces the alphabet size of the syntax element and allows more efficient operation of *context modeling* stage.

There are four main binarization scheme employed in this stage: unary (U) binarizer, truncated unary (TU) binarizer, k-th order exp-Golomb (EGk) binarizer and fixed-length (FL) binarizer. Each one of them applies a different mathematical transformation to the syntax element. Also, two combinations of these binarization schemes are used for some syntax elements: FL+TU, TU+EGk (also called UEGk).

**2) Context modeler** is the heart of context-adaptive capability of CABAC that differentiate it from other entropy coding techniques. It assigns a model probability distribution to given symbols which are used for generating the code at the subsequent coding stage. This model determines the code itself and controls the efficiency of the coding. It is kept up-to-date at all times meaning its statistics is updated after coding of every new bit. It consists of a table of 399 entries which each consists of a 6-bit probability value and a 1-bit MPS (Most Probable Symbol). The table is accessed and updated by binary arithmetic coding stage; it is initialized with some predefined values (3 variations of initial table exist that depending on encoding parameters one is selected for an encoding scenario) at the beginning of each slice.

**3) Binary arithmetic coder** is another differentiating feature of CABAC. It is based on recursive interval subdivision. The interval and its location are tracked at any time by two integer values. Based on the statistical property of the symbol being coded, the interval is divided to two regions proportional to probability of LPS and MPS. Update of this interval produces 0 or more output bits to be appended to the output stream. A context-model entry (associated with the symbol) provides the statistics of the symbol being coded. The 6-bit of context entry is the probability estimate of the Least Probable Symbol (LPS) while the 1-bit of the entry shows the polarity of MPS.

Since the sub-interval size is reduced after each coding, a renormaliztion operation rescales the interval range and location to proper range. Actually this renormaliztion process generates the output bit as part of its rescaling process.

**4) Bypass coding** is a simplified form of arithmetic coding applied to symbols that more or less show a uniform distribution so statistics doesn't help to improve their coding efficiency. As the result, their coding doesn't need to reference or update the context modeler table. This method also uses the same interval subdivision and renormalization scheme.

Now the use-case specification for CABAC as a whole can be described as below:

Name: **UC1)** High-level use-case of CABAC
Description: CABAC receives a syntax element from the higher level code of H. 264 encoder (namely Slice Generator) and entropy encodes it.

Precondition(s): A valid syntax element (MB, sub-MB, header element, type, …) is passed to CABAC.

Postconditions(s): CABAC state is updated based on its previous state and the new syntax element. Some output bits might be generated.

- Basic Course of Action:

1. CABAC receives a syntax element from a variety of possible syntax element sources within slice generator
2. It binarizes the syntax element through one or combination of *unary, truncated unary, k-order exp-Golomb and fixed-length* methods into bins.
3. It fetches a proper context entry from the context modeler for each bin based on history, bin index, …
4. Using the statistical info of the context entry, it encodes the bin using binary arithmetic coder.
5. It updates the context entry based on the bin encoded.
6. It updates the arithmetic coder state.
7. It generates 0 to potentially several bits to be written to output stream.

- Alternative course A: Bypass coding mode is used for syntax elements with uniform probability distribution. This replaces steps 3-7 above.

2. It passes the bin along the coder state to a bypass coder.
3. It updates the arithmetic coder state.
4. It generates 0 to potentially several bits to be written to output stream.

And its use case diagram showing its main stages is as below:

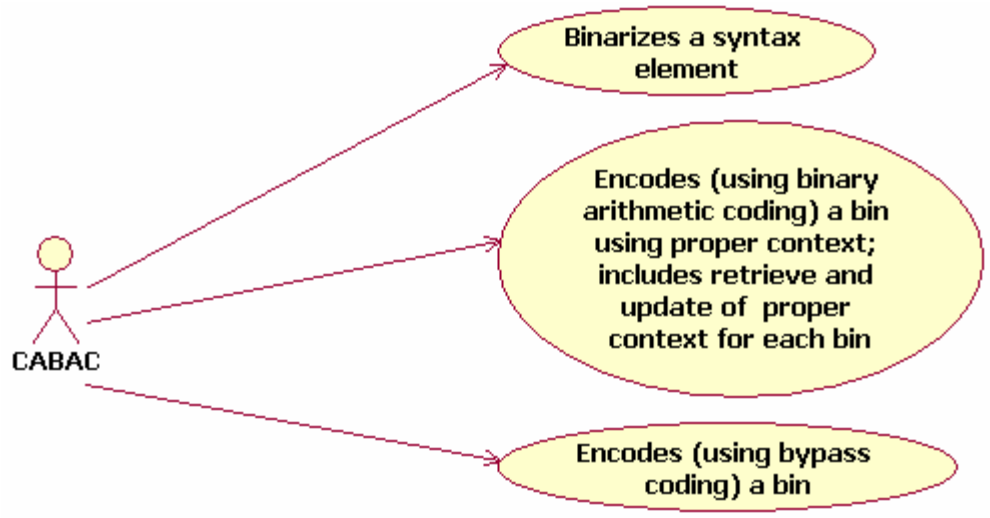


Figure 7. High-level CABAC use-case

# 4. Analysis and design of an OO CABAC

Because of previous involvement with CABAC, it wasn't too difficult to come up with the set of initial class candidates. Below figure was one the first thoughts.

But after a while and also considering higher level issues (e.g. interface of CABAC within the whole reference software), the pool of class name started to evolve significantly and the changes continued (though with slower rate) till figuring out the detailed interaction sequence after each iteration of refinement.

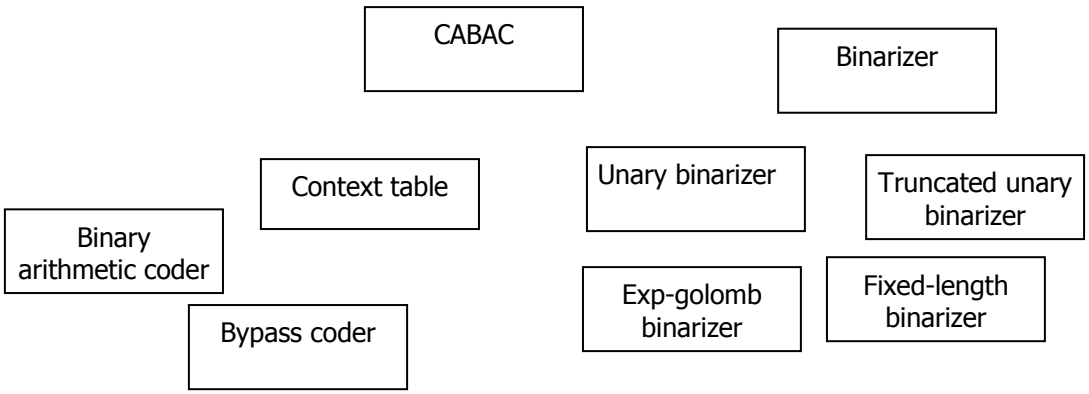


Figure 8. Initial pool of CABAC's class names

A natural decision was to completely separate entropy coding (as a whole) from the rest of H.264 coder through a clean interface (no matter if the final implementation of CABAC was a component or just simply a package). A weak point in the reference software implementation was having an if-else block for every access to CABAC/CAVLC to setup the different settings required for each one (though through using function pointers it was attempting to use a single function call). Using a base class for entropy coding with different generalizations as CABAC and CAVLC was sounding like a good decision. The base class itself was realization of an interface so implementing CABAC and CAVLC as components was becoming easy. Then higher level portion of H.264 would use a single interface to access entropy coding services. This interface is retrieved by searching and loading of the right component (CABAC/CAVLC e.g. through MS COM component enquiry APIs) based on whether the codec supports the required profile and whether if the right component is found. Note that in a real life codec the entropy coder is an integral part of the codec and we here assume it as a component but the argument of a clean and working interface applies to both component or package cases. At the end, higher level portion of the codec seamlessly uses the interface no matter what mode of entropy coding is used on the underlying layer (CABAC vs. CAVLC).

Figure 9 shows how CABAC and CAVLC are realizing IEntropyEncoder interface. The rest of codec only sees this interface from these components. The main method of this interface is EncodeElement which receives a SyntaxElement

object. This object contains type and value of the syntax element to be entropy coded. CABAC/CAVLC looks into type member of this object to rout it to the proper handler. Basically EncodeElement behaves like a dispatcher. But entropy coder component also need to know about some data (all related to Macroblock properties) kept by codec. Though theoretically all of these data can be retrieved and passed to the EncodeElement as a big data structure, this would shift some knowledge of entropy coder to the caller (in codec) as not all of this data is necessary for each syntax element. This breaks our goal of good decomposition. A better approach tries to expose an interface from the caller (a Macroblock object) to entropy coder component so then CABAC/CAVLC only query the data they need at the right time so this separates the behavior in a good fashion.

IMacroblock interface only exposes the data that CABAC/CAVLC might need to enquire about it. Note that this data access is read only so coupling is not an issue. Also note that IMacroblock belongs to the Macroblock object and not kept for a long time (unlike IEntropyEncoder which could be kept by codec for the whole lifetime of codec) that entropy encoding is to be done for some of its elements so the next Macroblock object would pass a different interface reflecting its own identity down to EncodeElement.

The rest of classes and enumeration types are the types that used by *both* layers (entropy coding layer and codec layer), e.g. SyntaxElement that is created by codec layer and passed down to entropy coding layer (read only access). Bitstream object is passed to entropy codec layer (through AssignBitstream) at start of each new slice to inform the entropy coder component about the target buffer it needs to store the generated encoded stream into, so this is the only scenario that entropy coder has a side effect on codec layer.

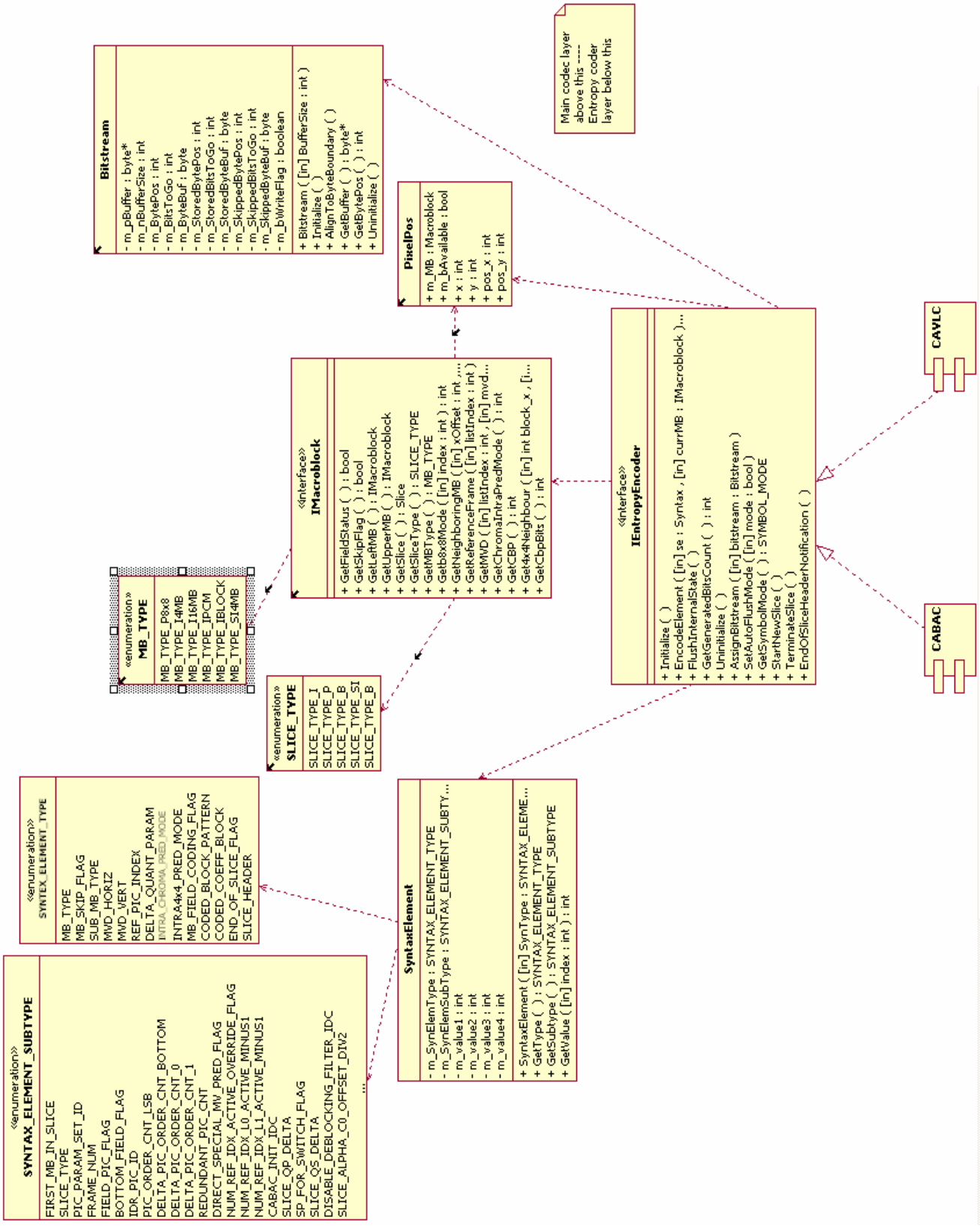


Figure 9. Interface of CABAC/CAVLC with rest of codec (InterfaceLayer diagram in model).

Figure 10 shows both CABAC and CAVLC class diagrams. They both derive from BaseCoder class which implements the basic functionality of the entropy coder. It implements general administrative methods like EncodeElement and AssignBitstream but CABAC and CAVLC overrides the rest of the methods based on their own encoding logic. Since both CABAC and CAVLC exactly use the same mathematical transformations (members of VLC_Binarizer) to encode the slice header elements, implementation of EncodeSliceHeaderSubSyntax() is done in the base class.

CABAC is significantly larger than CAVLC. It isn't broken to separate packages as all of its classes are pretty much related (though was tempted to do it for Context-related classes). The reference software had integrated binarization and context retrieval. Here, they're separated. There was no need to provide multiple binarizer classes as they implemented as simple transformation methods without keeping state so ended up as methods of a single static class of CABAC_Binarizer. Since both binary coder and bypass coder were sharing the same sub-interval properties, it was decided to merge their classes (from the initial pool of class candidates) as BinaryArithmeticCoder and have a method for bypass coding.

ContextModeler turned out to be much more challenging than originally thought (since binarization was separated from context management). Basically its function is to retrieve the proper context entry for a bin. For most syntax elements only knowing the index of bin (within a bin string) being encoded is enough to make decision about the right context entry (of course based on syntax type, …). So by calling StartElementEncoding at start of encode of a syntax element and saving its syntax element and IMacroblock interface temporarily till end of encode of that element (notified by calling FinishElementEncoding) and multiple calls in between to GetContextElement with increasing bin index, all context entries can be retrieved. But MB_TYPE and SUB_MB_TYPE follow a different structure (based on some code trees) so the whole tree is created right after StartElementEncoding(), its result stored temporarily in m_CodeTree_ContextIndex and retrieved through GetContextElement_Unstructured().

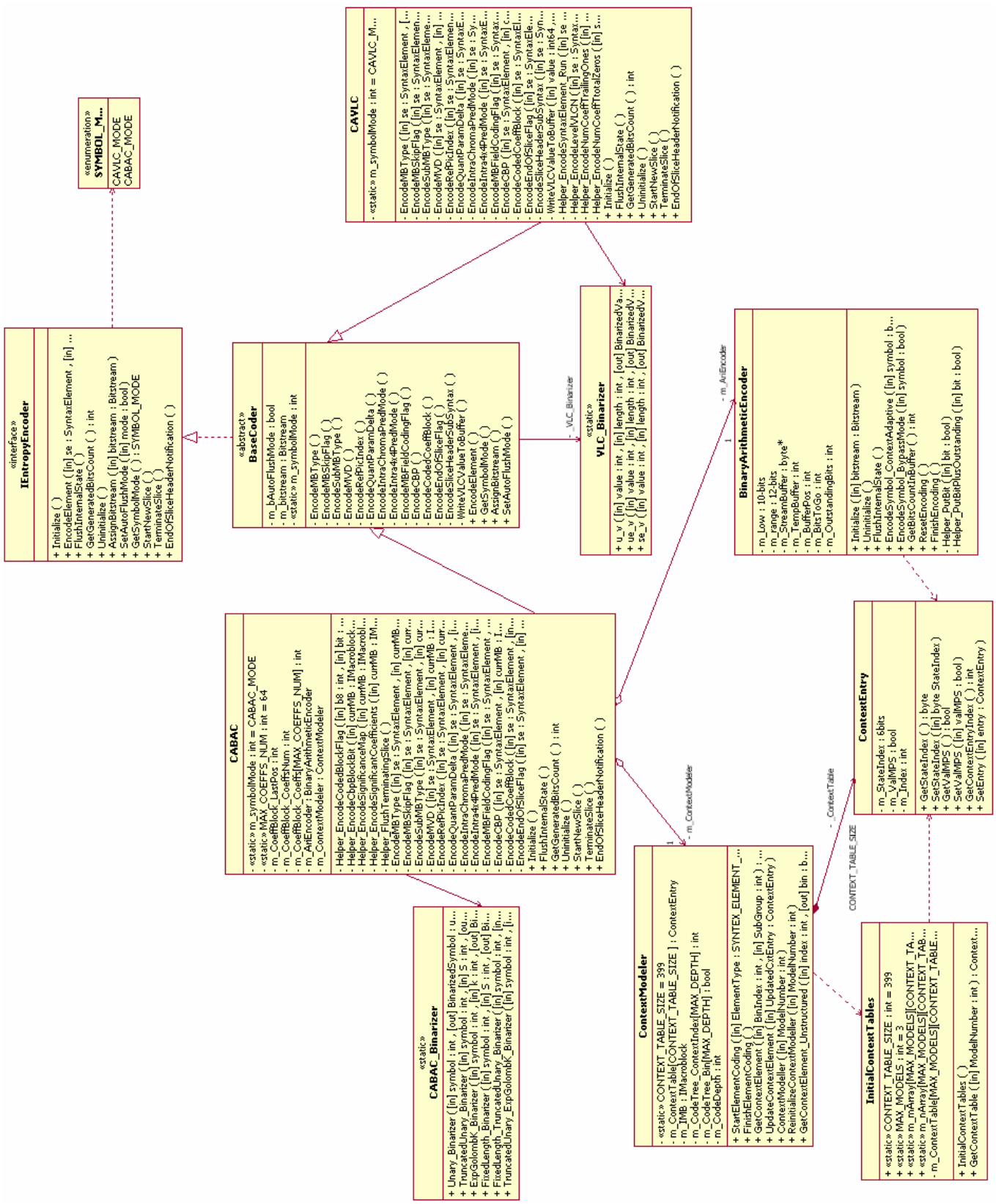


Figure 10. Entropy coder class diagram (both CABAC/CAVLC) (EntropyCoder_Slim class diagram in the model).

ContextEntry class simply keeps a single entry of the context table described before (7 bits in total) and provides method to retrieve and update its data. ContextModeler keeps CONTEXT_TABLE_SIZE of context entries. This table is initialized to some predefined values at start of each slice (when codec sends StartNewSlice notification). Then the content of table needs to be filled by one of the three predefined possible tables (selected based on model number). But actually the table itself is not predefined but can be calculated through predefined arrays of m_mArray and m_nArray. So to do only this calculation once, the constructor calculates the table once and then table content is retrieved by GetContextTable() at start of every new slice.

BinaryArithmeticEncoder class's main methods are EncodeSymbol_ContextAdaptive() and EncodeSymbol_BypassCoding(). One expects a ContextEntry while the bypass mode doesn't need one. These methods update the subinterval and generate the output bitstream in the Bitstream buffer provided by the codec.

Appendix A explains all the parameters need to be setup by the codec in an instance of SyntaxElement object to be passed to EncodeElement (through value1 to value4 members). The number of parameters varies from one to four depending on the syntax/sub-syntax type.

Appendix B explains the macroblock data the ContextModeler looks them up through IMacroblock interface and how it generates the context index increment (CxtIncr) to be added to the base group index of the syntax element or the final CxtIndex directly.

For more details on each method please refer to each method's documentation in EntropyCoder diagram of the Rose model. This model shows flat relationship of entropy coder layer and codec layer (not encompassing individual components, unlike InterfaceLayer and EntropyCoder_slim diagrams). The classes and their methods were refined several times iteratively after test-driving the main use-cases and figuring-out issues.

Figure 11 shows a more-detailed use-case diagram of CABAC as a whole but from initiation point of view from codec layer (slice generator). UC2 use-case specification shows the sequence of a sample syntax element encoding here for a MVD (motion vector difference) type. Other syntax elements would require slightly different specifications. And Figure 12 shows the interaction diagram of similar element encoding for MVD.

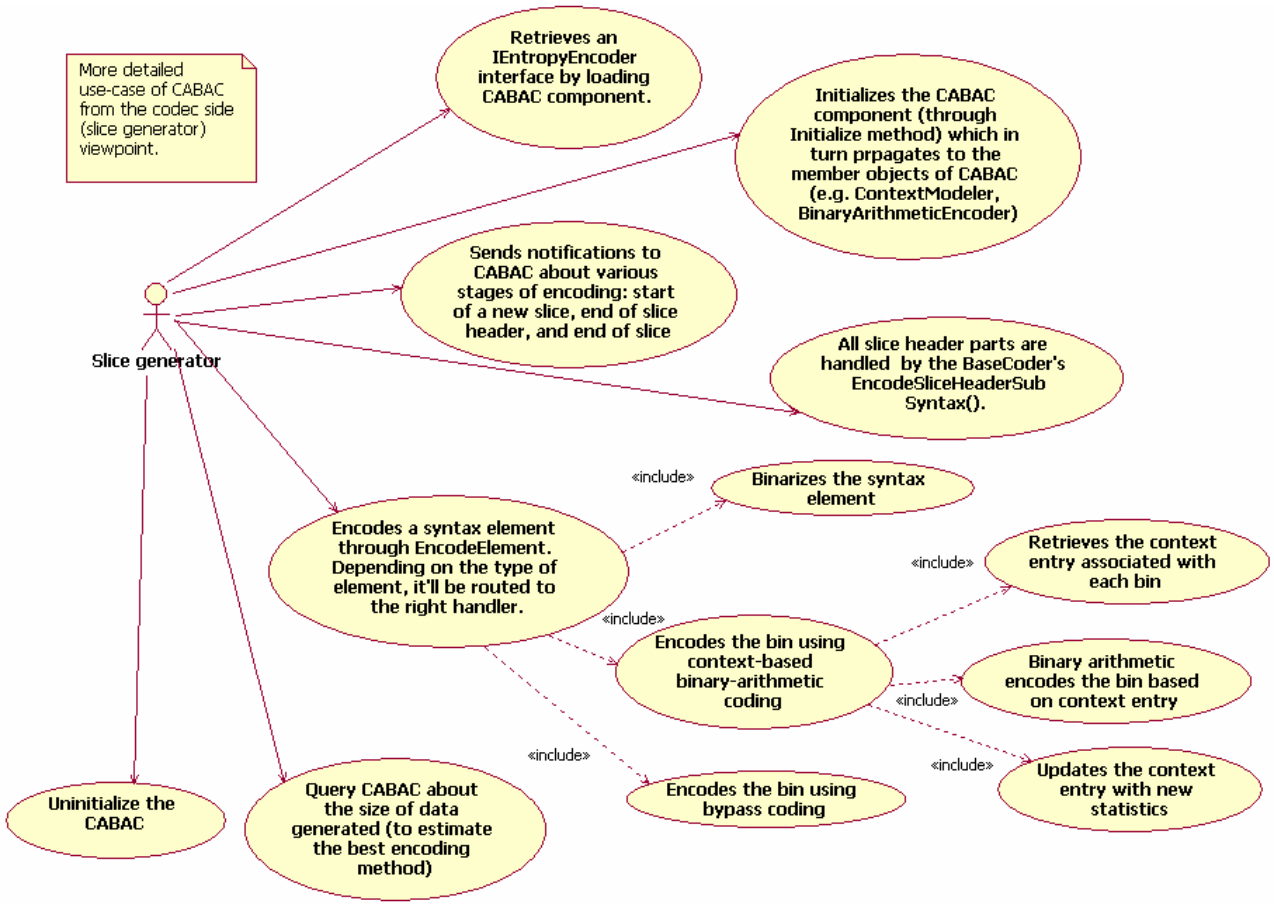


Figure 11. Detailed CABAC use-case diagram from codec viewpoint

Name: **UC2)** Encode of a sample syntax element use-case
Description: An example of entropy coding of a syntax element (not of slice header type), for example of MVD type.

Precondition(s): Codec layer has prepared a SyntaxElement instance identifying the type and value of the element to be encoded. An instance of the CABAC is already retrieved (through retrieval of IEntropyEncoder interface) and initialized. A slice is already started and its header already encoded.

Postconditions(s): CABAC state (sub-interval info) is updated based on its previous state and the new syntax element encoded. Some new output bits might have been generated and appended to the bitstream buffer.

- Basic Course of Action:

1. CABAC receives a syntax element from a variety of possible syntax element sources within slice generator.
2. It routes it to the proper handler (e.g. EncodeMVD in this scenario). The rest of steps actually happen within the handler.

3. It binarizes the syntax element through one or combination of *unary, truncated unary, k-order exp-Golomb and fixed-length* methods into bins (in this scenario could be concatenation of a single bin, unary, 9-order exp-Golomb and few other bins.
4. It notifies the context modeler to prepare itself for retrieval of context entries for the new syntax element. The context modeler saves syntax element and Macroblock callback interface temporarily.
5. Now for each bin that needs a corresponding context entry, it queries context modeler to receive the proper context entry.
6. For the context-dependent bins, it encodes the bin by passing the bin and the context to arithmetic encoder.
7. It updates the context modeler's context entry by passing it the updated context entry returned through arithmetic encoder.
8. For the context-independent bins, it uses bypass coding instead.
9. It notifies the context modeler that encode of current syntax element is finished so it can releases its copy of Macroblock interface.
10. At the end of this process, sub-interval state of arithmetic coder is updated and 0 to potentially several bits are generated to be written to output stream.

- Alternative course A: For syntax elements of type MB_TYPE and SUB_MB_TYPE, a code-tree is used for context entry retrieval. This replaces above the step 5 with:
  5. It updates the arithmetic coder state. Now for each bin that needs a corresponding context entry, it queries context modeler to receive the context entry. The code tree was already prepared and stored in response to earlier preparation for context retrieval.

Figure 13 shows sequence diagram for CABAC first time initialization, and also later initialization/notification sequence at every start of new slice.

Figure 14 to 16 show state-chart diagrams for CABAC, ContextModeler and BinaryArithmeticEncoder classes respectively. And Figure 17 shows the use-case activity diagram for EncodeElement sequence already discussed.

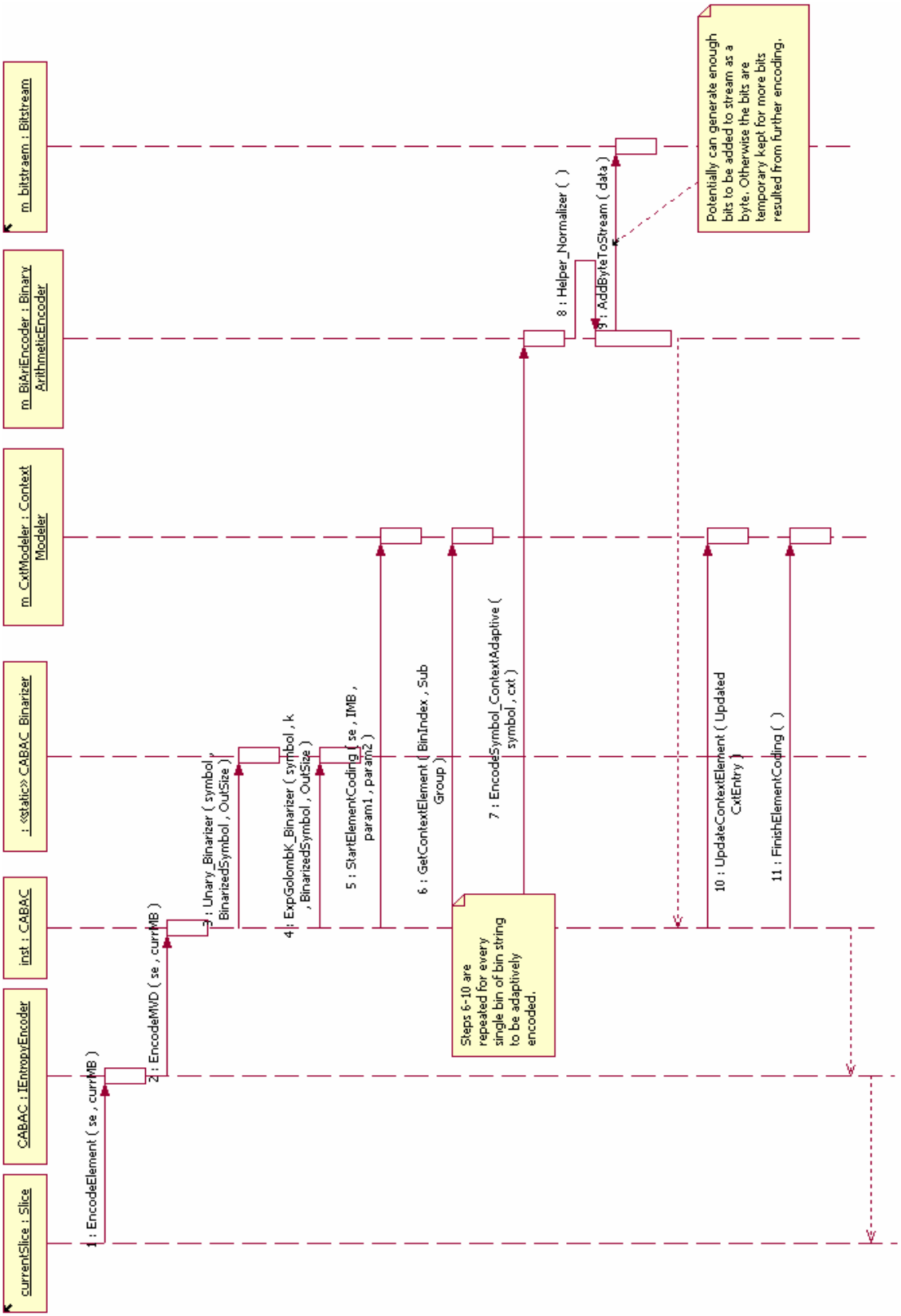


Figure 12. Sequence diagram of a sample syntax element coding for MVD element (SyntaxEncoding diagram in model).

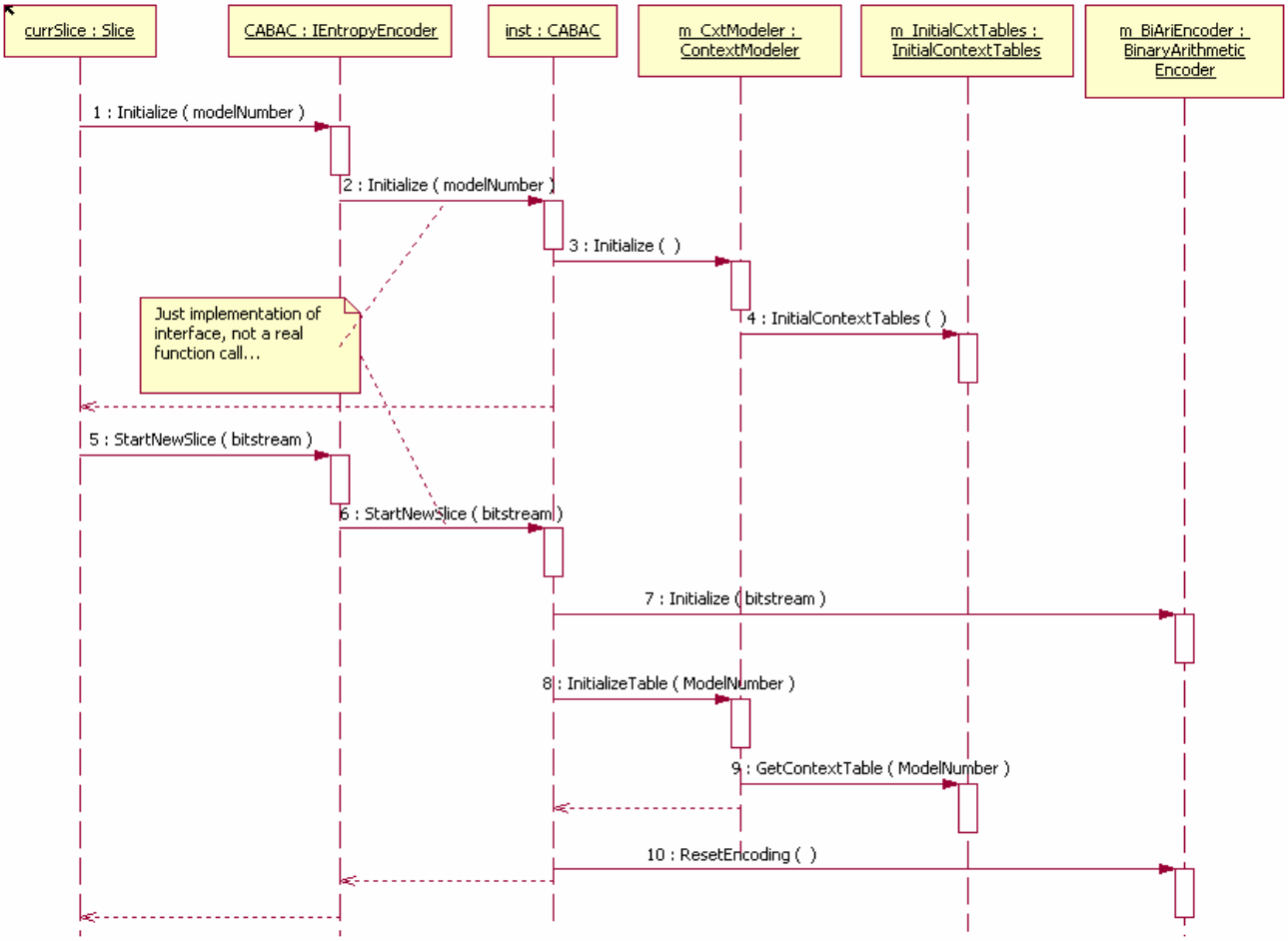


Figure 13. Sequence diagram of CABAC's first-time and later slice-time initializations (CABAC_Initialization diagram in model).

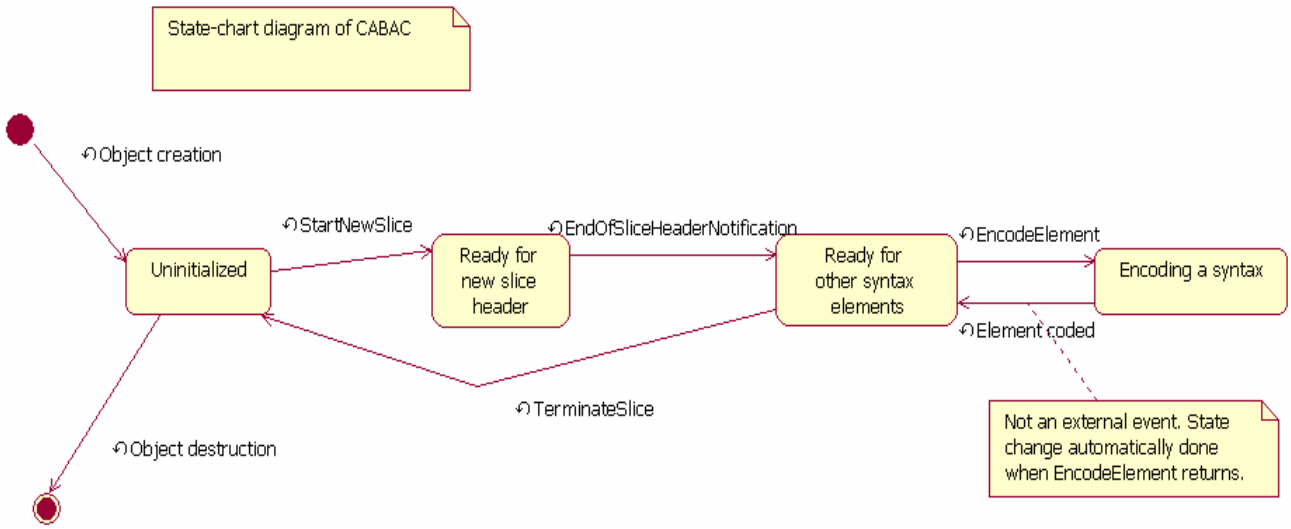


Figure 14. CABAC's state-chart diagram (CABAC state machine in model).

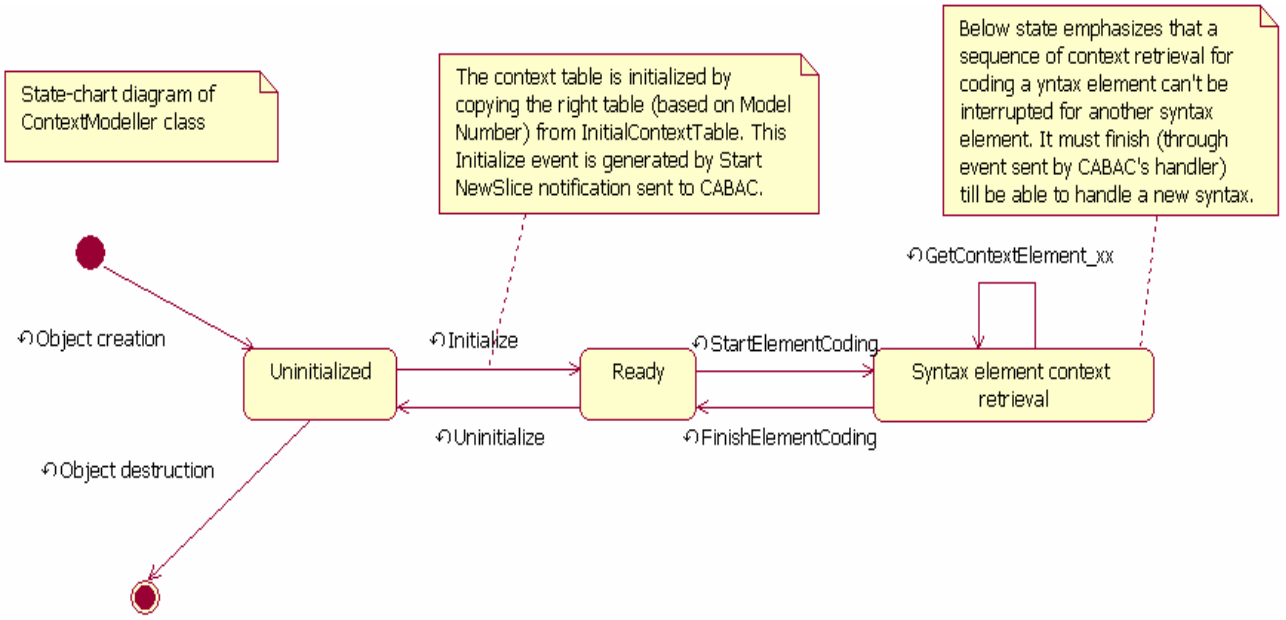


Figure 15. ContextModeler's state-chart diagram (ContextModeler state machine in model).

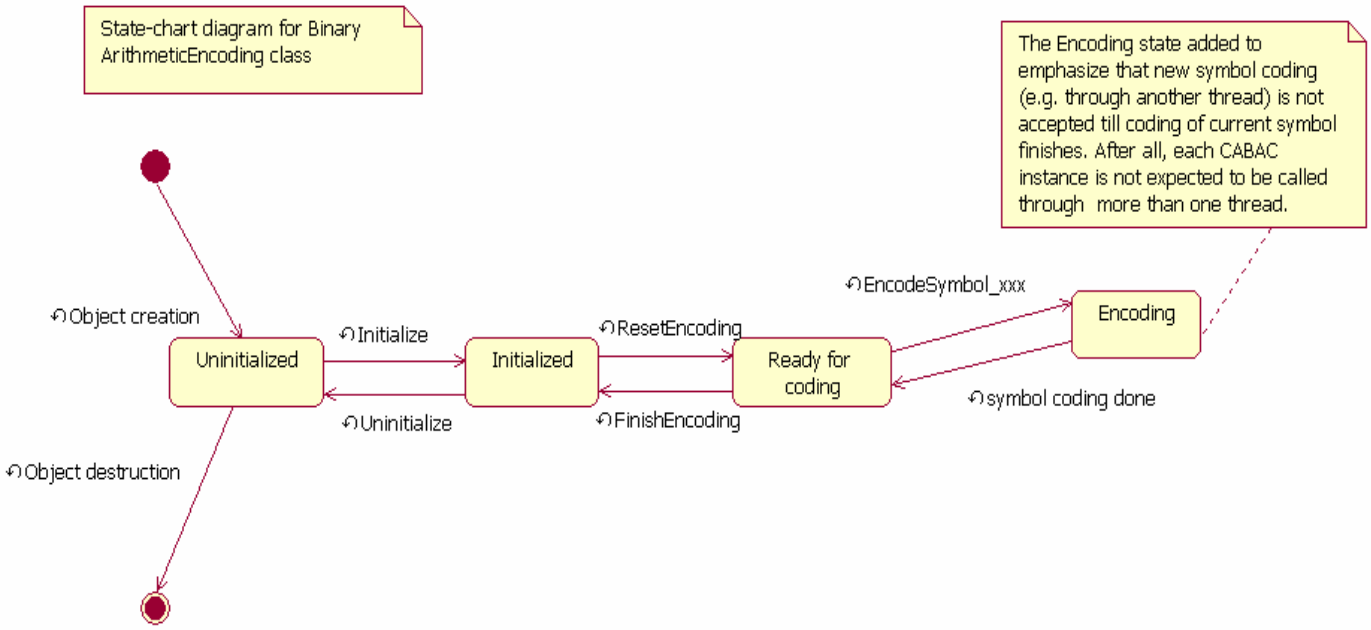


Figure 16. BinaryArithmeticEncoder's state-chart diagram (BinaryArithmeticCoder state machine in model).

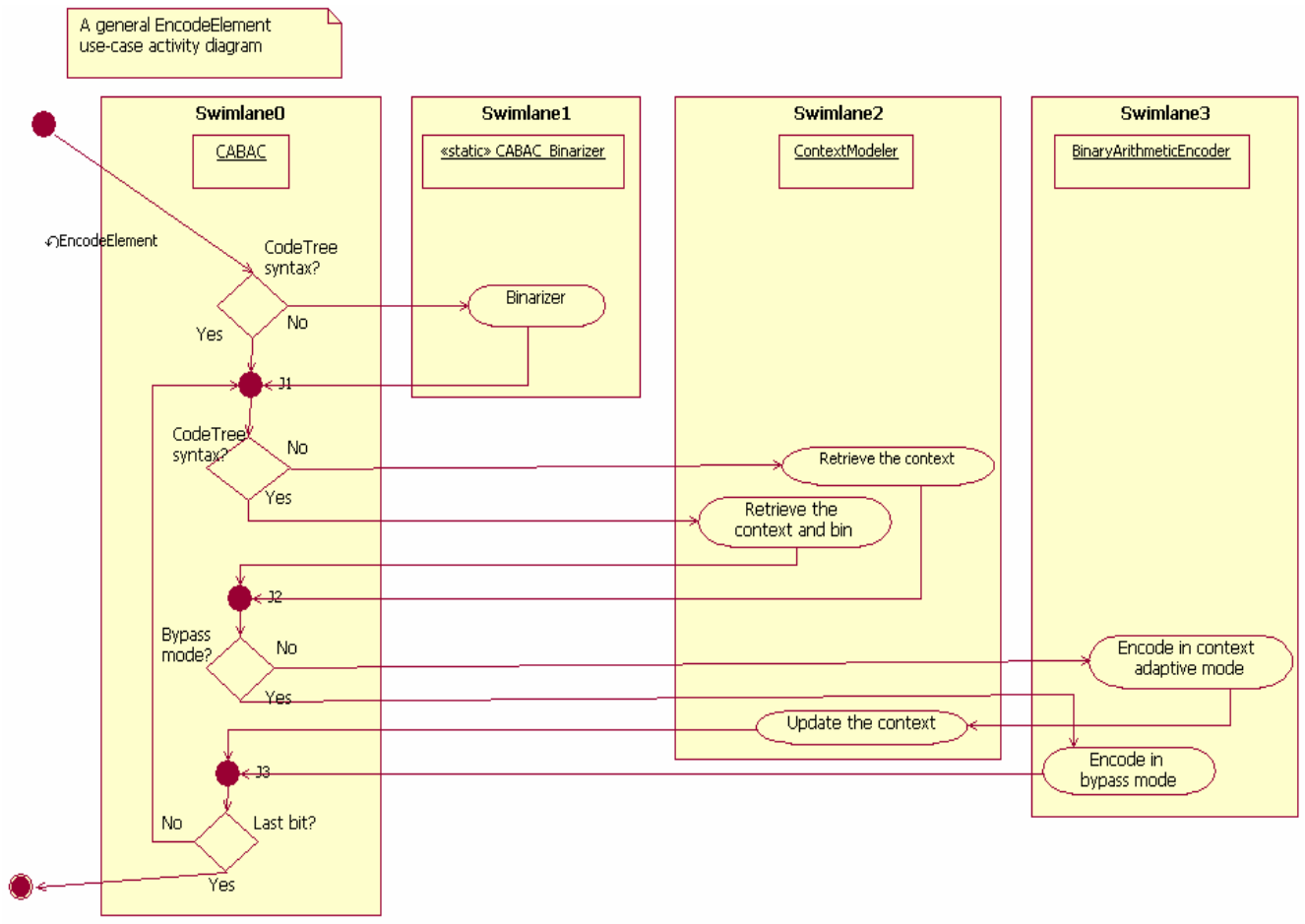


Figure 17. A generic EncodeElement use-case activity diagram (EncodeElement activity diagram in model).

# 5. Higher-level H.264 model

This portion only tries to decipher some portions of the higher-level codes of the H.264 codec get involved before entropy coding. As suggested before, the focus is at the slice layer and below so we ignore higher level administrative portion like encoding parameter parsing and setup, decision of slice type, … This is not an attempt to introduce a full-featured design but first to understand and analyze portions of the code which till now was considered horrifying for the author and at the same time trying to suggest some level of organization (classes/packages) to encapsulate that code. Needless to say, this reverse-engineering of the code was accompanied by lots of debugging and comparing the code against the standard and other books in the area.

Below use-cases UC3 & UC5 describe the use-case specification of encoding an intra/inter slice respectively. These use-cases are the main use-cases of the codec that derive the majority of codec operations. Because of their complexity, a layering of use-cases needs to be done, e.g. including UC4 (RD cost calculation for a macroblock) in UC3. Similar thing needs to be done for most of the steps of UC3 and UC5 steps.

Figure 18 shows the high-level layering of classes, packages and components in the codec scenario. For a full description of the main class diagrams of the codec (excluding entropy coder) refer to Slice class diagram of the model. It was too big to include here. And most of the time was spent to derive this class diagram.

Figure 19 shows a high-level sequence diagram for encoding a new slice without going to all the huge details and interactions. While Figure 20 shows the activity diagram for RD_Cost_For_Macroblock method of the Rate Distortion Manager class.

Name: **UC3)** Encoding a new **intra (I)** slice
Description: Encode of a new intra slice (e.g. a frame). [It's mainly what happens through encode_one_slice()]

Precondition(s): Codec is in a valid state (i.e. previous frames encoded properly). And a new uncompressed frame is provided to the codec.

Postconditions(s): The frame is encoded according to the standard spec (i.e. its generated bitstream is compliant with the spec.)

- Basic Course of Action:

1. It initializes the slice state including reset of the basic slice data, allocation of required memory, reset of the reference picture lists,…

2. It writes the slice header to the output bitstream buffer and adds the extra bits to make it byte-aligned. Also notifies the entropy encoder that header is written.
3. Now it applies all steps 4-8 below for every single macroblock of the current frame in the scanning order (from top-left corner to bottom-right corner). [done through calling encode_one_macroblock]
4. It calculates all possible chroma prediction modes (DC, horizontal, vertical and plane) for both U and V components and stores their result temporarily.
5. It picks one of the above chroma prediction modes and tries both possible luma prediction modes (Intra16x16, Intra4x4) by repeating the 2 steps below.
6. It calculates the RD cost of picking this luma/chroma mode setting through use-case **UC4**. [equiv. to RD_cost_for_macroblock() in code]
7. If the resulted cost is the minimum cost so far, the setting and resulted predictions/residuals are saved temporarily. Goes back to step 5 once to try the other luma prediction mode.
8. Now that the best choice of setting is found, entropy encodes the result and writes the result to the bitstream. Goes back to step 3 for the next macroblock.
9. Now that all macroblocks of the slice are encoded, it terminates the slice.
10. The final bitstream is NAL processed and flushed to the output file.

Name: **UC4)** Calculating cost of encode of a macroblock using a selected luma/chroma mode
Description: This use-case calculates the cost (considering both distortion and entropy-coded rate) of encoding a macroblock using a particular pair of luma/chroma prediction mode. [equivalent to RD_Cost_for_macroblock()]

Precondition(s): Codec is in a valid state. All chroma prediction modes are already calculated and stored temporarily.

Postconditions(s): The state of encoder is not changed (really changed e.g. for entropy coder but restored to the original state after end of the use-case).

- Basic Course of Action:

1. For all possible prediction modes of the above luma intra mode (DC, Hor, Vert, Plane for I16x16; all 9 modes for I4x4), DCT transformation of the modes calculated and only the one with the least SAD selected.
2. It also calculates the DCT transformation of the residual resulted from the selected chroma mode.
3. It calculates the quality distortion of both luma and chroma components (loss).
4. It calculates the bit count needed if with he above luma/chroma prediction modes the whole macroblock was entropy coded.
5. It assigns a cost value to this selected luma/chroma settings based on calculated distortion/rate and using a cost formula.

Name: **UC5)** Encoding a new **inter (P/B)** slice
Description: Encode of a new inter slice (e.g. a frame). [It's mainly what happens through encode_one_slice()]

Precondition(s): Codec is in a valid state (i.e. previous frames encoded properly). And a new uncompressed frame is provided to the codec for inter coding.

Postconditions(s): The frame is encoded according to the standard spec (i.e. its generated bitstream is compliant with the spec.)

- Basic Course of Action:

1. It initializes the slice state including reset of the basic slice data, allocation of required memory, reset of the reference picture lists,…
2. It writes the slice header to the output bitstream buffer and adds the extra bits to make it byte-aligned. Also notifies the entropy encoder that header is written.
3. Now it applies all steps 4-8 below for every single macroblock of the current frame in the scanning order (from top-left corner to bottom-right corner). [done through calling encode_one_macroblock]
4. For all the three cases of non-8x8 motion search modes (namely 16x16, 8x16, 16x8), it repeats steps 5-6.
5. It motion estimates the block using the selected mode.
6. If there are multiple reference frames possible (depending on forward/backward lists), the best one is selected.
7. It calculates the cost of blocks motion estimated (one block for 16x16 case and two blocks for each of 8x16 and 6x18 cases) and store it.
8. Now the focus goes to the case of four 8x8-blocks. For each 8x8 block steps 9-13 is repeated.
9. It picks one of the 4 possible (5 for B slice), sub-partition modes (8x8, 8x4, 4x8, 4x4) and repeats step x-y below.
10. It Motion estimates the block using the sub-partition mode selected.
11. If there are multiple reference frames possible (depending on forward/backward lists), the best one is selected.
12. It calculates the cost of encoding that 8x8 block using the selected sub-partition mode.
13. It selects the best sub-partition mode resulting in the least cost for that block. It goes back to step 10 to repeats this for the other block.
14. By adding up the cost of each 8x8 block (each using the best sub-partition mode), now the best cost for the four possible block partitioning is known (16x16, 16x8, 8x16, 8x8) so the best one is chosen.
15. Similar to steps 4-8 of the **Intra** use-case, now the macroblock is intra-coded and the best possible mode for luma/chroma is selected.
16. When reaching this point, the best possible mode from all inter/intra cases is selected.
17. Now that the best choice of setting is found, it entropy encodes the result and writes the result to the bitstream. Goes back to step 3 for the next macroblock.
18. Now that all macroblocks of the slice are encoded, it terminates the slice.
19. The final bitstream is NAL processed and flushed to the output file.

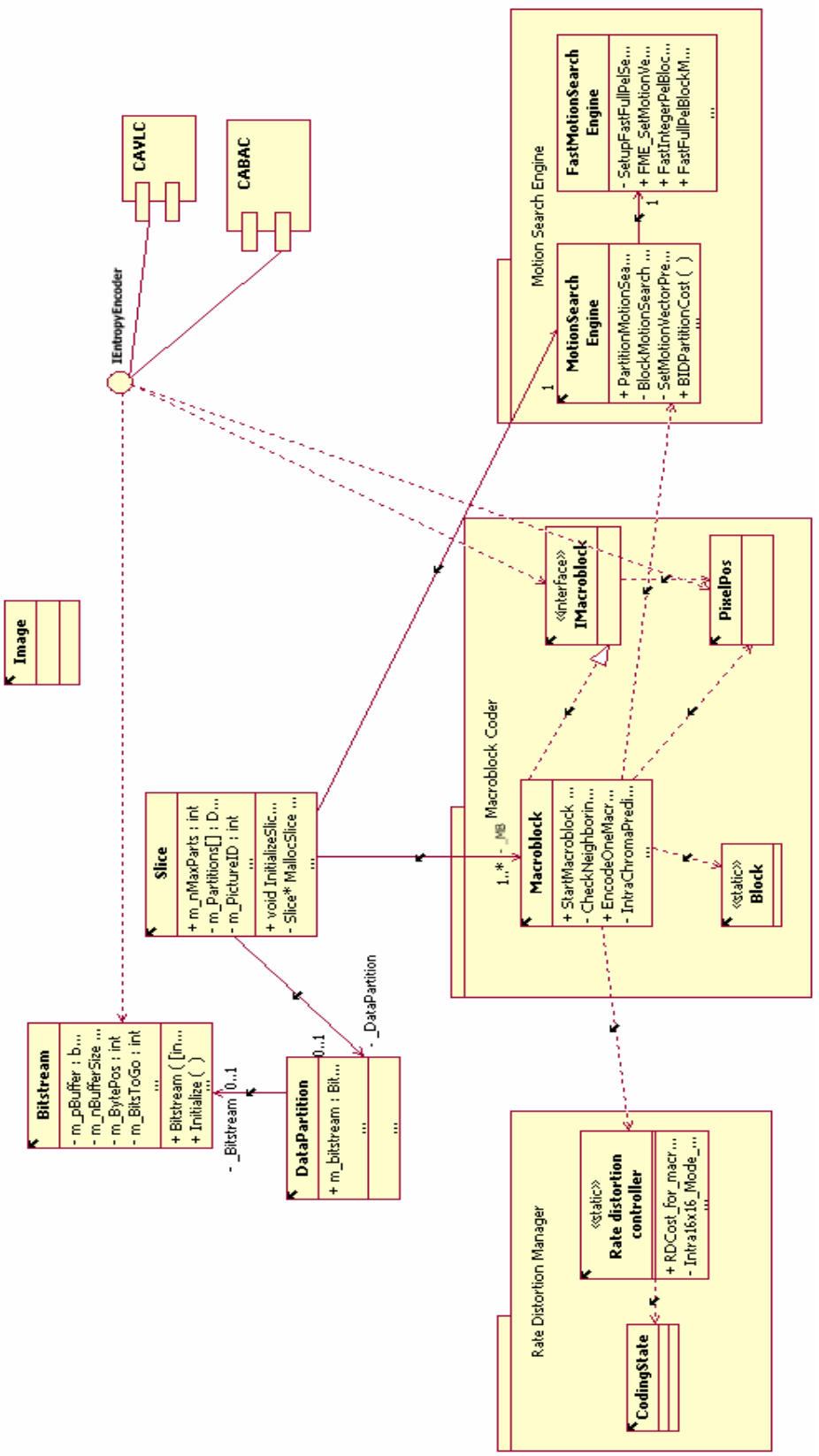


Figure 18. High-level relationship of class/package/components of the whole codec (Codec diagram in model).

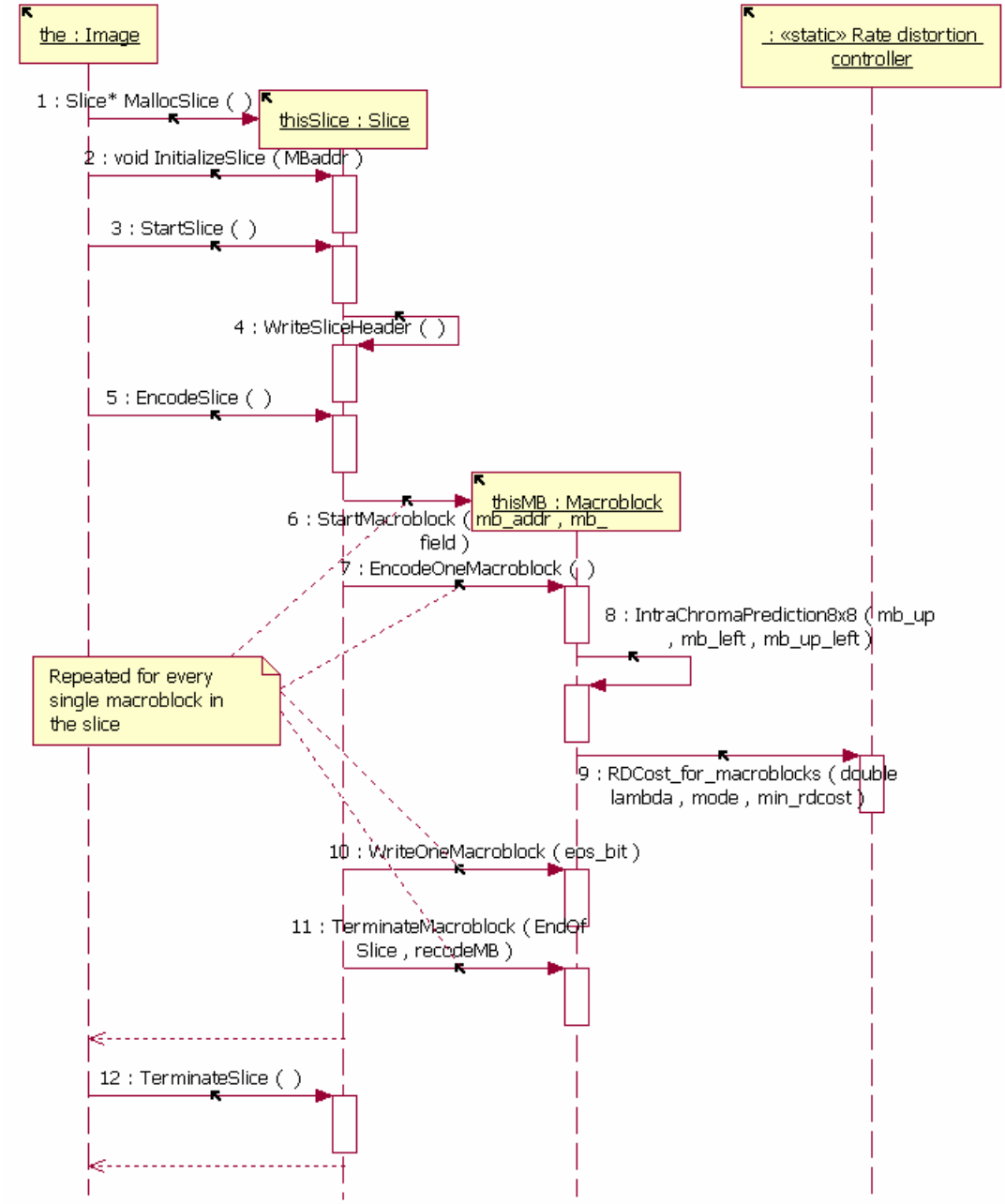


Figure 19. A brief sequence diagram for "Encoding a new Intra slice" use-case (EncodeOneSlice sequence diagram in model).

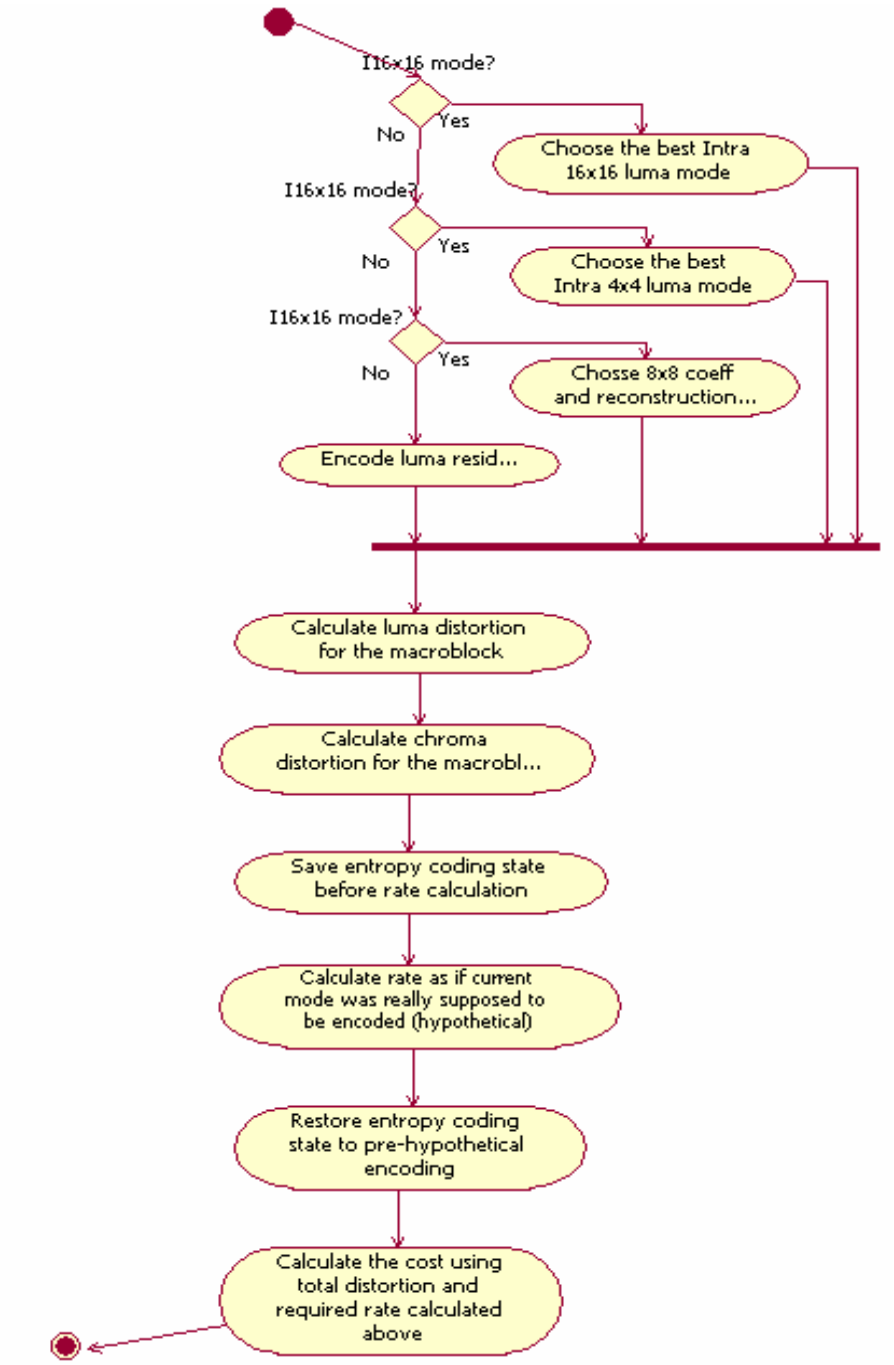


Figure 20. RD_Cost_For_Macroblock operation activity diagram; part of Rate Distortion Controller class (RD_CostForMB diagram in model).

# 6. Conclusion

## Outstanding issues:

The CABAC/CAVLC components analysis and design are pretty much done and no more holes are found. Several iterations of test-driving use-cases/sequence diagrams and update of classes and interfaces really helped to identify the holes and refine the design.

But same thing can't be said about the higher-level H.264 codec as it was originally predicted. Though some designs are produced, the analysis portion is not finished and there exists holes in this preliminary design (e.g. coupling between packages, …). Lots of time was spent on analysis of the existing code and figuring out the relationship between different entities but still some portion (though minor) of the code is untouched. The derived class diagrams shed some light on the relationship between classes, packages and generally how the layering of the codec can be done. Especially the IMacroblock interface cleanly isolates the access from entropy coder to higher-level codec functionalities.

There are still many other use-cases and sequence diagrams to be tested to help refine the classes and come with a complete design. At the same time, task of deciphering most of the daunting portions of the code is done. Another observation is that the Macroblock class has grown too much and should be broken to multiple classes. Most of the class methods are the counterparts of existing functions in the original code that are grouped under different classes. Some of these functions even go above 1000 lines (e.g. encode_one_macroblock) which need to be broken done.

## Deployment

The deployment of the system is straight-forward as there is only a single node where the H.264 encoder is running on and CABAC though is a separate component (of course a registration and enquiry mechanism is required like Microsoft's COM), it is an in-process component (e.g. running as a DLL within the encoder's process). Because of the large amount of data to be shared and low-degree of parallelism, a distributed system is not an option at this point though it is the focus of some existing research. But there have been successful attempts to use SMP (shared-memory multiprocessor) systems for H.264 encoding. But only limited portion of the code is parallelizable. For example, CABAC has completely a serial nature but rate-distortion cost calculation (to pick the best possible encoding modes from a pool of possible choices) or motion-estimation (to find a matching macroblock in other frames) can be parallelized.

Note that inter-process communication is too costly for the encoder scenario so even in the parallel form multithreading will be used. The current implementation of reference software codec does not support multithreading as it is not geared for performance and subsequently we have not considered here. Though CABAC

is hardly parallelizable, in the concept of parallel rate-distortion calculation, speculative entropy-encoding is an option. In this scenario several instance of CABAC component could be running in multiple threads each following a separate speculative path of rate-distortion algorithm. As CABAC is not using any global variables, having multiple instances each running in different thread is not an issue, but calling a single instance from multiple threads is an illegal action as it is against the serial nature of CABAC and can't gain anything. Though currently not enforced, a simple lock at the BaseCoder can safe-guard against this illegal action (since the whole CABAC is accessed through IEntropyEncoder interface, there are only few methods to be modified for grabbing/releasing the lock).

## Summary:

This project was an interesting experience to practice most of the OO analysis/design topics learned in class. This topic didn't have a clean metaphor to compare against as it was mainly of engineering/mathematical nature. The main interaction between different layers resembled client/server behavior though some elements were client of one layer while serving another layer.

Also, the importance of creating a good documentation especially a UML model of any software (especially large ones) couldn't be emphasized more as if this was the case for H.264 reference software, this project probably wouldn't exist at all!

## References:


- [Quatrani`03] Terry Quatrani, "Visual Modeling with Rational Rose 2002 and UML", Addison Weseley Professional, 2003.
- [ITU`03] ITU, "ITU-T Recommendation H.264: Advanced video coding for generic audiovisual services", 05/2003.
- [Marpe`03] D. Marpe, H. Schwarz, and T. Wiegend, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard" in *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 13, No. 7, July 2004.
- [Richardson`03] Iain Richardson, "H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia", John Wiley & Sons, 2003.

**Appendix A: SyntaxElement parameters EncodeElement() expects to receive for each syntax type/sub-type. Also showing dependency of the handler on other data (to be provided through IMacroblock)**.

| | value1 | value2 | Value3/4 | Dependences |
|---|---|---|---|---|
| MB_TYPE: | MBType2Value(currMB->mb_type) | | | Availability and mb_type of up/left neighboring MBs;<br>Slice_type (I/B/rest);<br>1-13 bins; |
| MB_SKIP_FLAG: | MBType2Value(currMB->mb_type) | currMB->cbp | | Availability and skip_flag of up/left neighboring MBs;<br>Slice_type (B/non-B);<br>Cbp; MB type;<br>1 bin only; |
| SUB_MB_TYPE: | B8Mode2Value (mode, pdir) [mode: partitioning mode, pdir: prediction direction] | | | Slice_type (B/non-B);<br>1-5 bins; |
| MVD: Includes MVD_HORIZ, MVD_VERT context entries | curr_mvd | 2*k+list_idx [identifies component/ direction] | subblock_x/y | Availability, mb_field of MBs containing neighboring sub-macroblocks;<br>MVD and position of neighboring sub-macroblocks;<br>Slice field/frame;<br>1 bin + UEGK(3) of value1-1 + bypass coding of sign |
| REF_PIC_INDEX: | ReferenceFrame | (fwd_flag)? LIST_0: LIST_1 | subblock_x/y | Availability, mb_field, sub-block mode of MBs containing neighboring sub-macroblocks;<br>Position of neighboring sub-macroblocks;<br>Slice_type (B/non-B);<br>Slice field/frame;<br>1 bin + unary coding of value1-1; |
| DELTA_QUANT_PARAM: | currMB->delta_qp | | | Last_dquant;<br>1 bin + unary coding of value1-1; |
| INTRA_CHROMA_PRED_MODE: | currMB->c_ipred_mode | | | Availability and c_ipred_mode of up/left neighboring MBs;<br>1 bin + TU(2) coding of value1-1; |
| INTRA_PRED_MODE: Includes PREV_INTRA_PRED_MODE_FLAG, REM_INTRA_PRED_MODE context entries | (mostProbableMode == ipmode) | Ipmode | | 1-4 bins; |
| MB_FIELD_CODING_FLAG: | currMB->mb_field | | | Availability and mb_field of A/B neighboring MBs; |

| | | | | |
|---|---|---|---|---|
| | | | | 1 bin only; |
| CODED_BLOCK_PATTERN:<br>CBPLuma = cbp % 16<br>CBPChroma = cbp / 16 | currMB->cbp | CBP_INTRA/<br>CBP_INTER | | Availability, mb_type and cbp of upper/left MBs;<br><br>availability, mb_type, position and cbp of MB containing left neighboring sub-macroblock;<br>Position of neighboring sub-macroblocks;<br>Slice_type (B/non-B);<br>Slice field/frame;<br>4 bins: 1 bin per each 4 luma blocks of cbp;<br>1 bin showing (chromaCbp !=0)<br>If true:<br>1 bin showing (chromaCbp ==2); |
| When cbp is present, CBPLuma specifies, for each of the four 8x8 luma blocks of MB, one of cases:<br>- All transform coeff levels of four 4x4 luma blocks in the 8x8 luma block are equal to zero<br>- One/more transform coeff levels of one/more of the 4x4 luma blocks in 8x8 luma block is non-0.<br>The meaning of CBPChroma is:<br>0 All chroma transform coeff levels are equal to 0.<br>1 One/more chroma DC transform coeff levels is non-0. All chroma AC transform coeff levels equal 0.<br>2 Zero/more chroma DC transform coeff levels is non-0. One/more chroma AC transform coeff levels are non-0. | | | | |
| CODED_COEFF_BLOCK:<br>Includes:<br>CODED_BLOCK_FLAG,<br>SIGNIFICANT_COEFF_FLAG,<br>LAST_SIGNIFICANT_COEFF_FLAG<br>COEFF_LEVEL | Level | Run | subblock_x/y;<br>block_type;<br>field/frame | Availability, mb_type and position of MB containing upper/left neighboring sub-macroblocks;<br>1 bin for cbp_flag;<br><br>For each coefficient, 1 bin to show significance of coefficient followed by another 1 bin to reflect last significant bit status;<br>For each significant coefficient traversed from end of the block:<br>- 1 bin to show if it's absolute value is equal to 1<br>- UExpG(14) coding of absolute value of coefficient-1<br>- 1 bin for sign of coefficient using bypass coding |
| - Depending on MbPartPredMode( mb_type, 0 ), the following applies.<br>- If MbPartPredMode( mb_type, 0 ) is equal to Intra_16x16, the transform coefficient levels are parsed into the list Intra16x16DCLevel and into the 16 lists Intra16x16ACLevel[ i ]. Intra16x16DCLevel contains the 16 transform coefficient levels of the DC transform coefficient levels for each 4x4 luma block. For each of the 16 4x4 luma blocks indexed by i = 0..15, the 15 AC transform coefficients levels of the i-th block are parsed into the i-th list Intra16x16ACLevel[ i ].<br>- Otherwise (MbPartPredMode( mb_type, 0 ) is not equal to Intra_16x16), for each of the 16 4x4 luma blocks indexed by i = 0..15, the 16 transform coefficient levels of the i-th block are parsed into the i-th list LumaLevel[i].<br>- For each chroma component, indexed by iCbCr=0..1, 4 DC transform coefficient levels of the 4x4 chroma blocks are parsed into iCbCr-th list ChromaDCLevel[ iCbCr ].<br>- For each of the 4x4 chroma blocks, indexed by i4x4 = 0..3, of each chroma component, indexed by iCbCr = 0..1, the 15 AC transform coefficient levels are parsed into the i4x4-th list of the iCbCr-th chroma component ChromaACLevel[ iCbCr ][ i4x4 ].<br>See page 46 of standard spec | | | | |
| END_OF_SLICE_FLAG: | EndOfSlice | | | 1 bin showing end of slice status (using context index 276) |
| | | | | |

| SLICE_HEADER: | | | | |
|---|---|---|---|---|
| | Sub-type | value1 | value2 | Coding |
| | FIRST_MB_IN_SLICE: | CurrrentMB_number | MbaffFrameFlag | ue(v) |
| | SLICE_TYPE: | SliceType | | ue(v) |
| | PIC_PARAM_SET_ID: | PicParamSet | | ue(v) |
| | FRAME_NUM: | CurrentSliceNumber | | u(v) |
| | FIELD_PIC_FLAG: | FieldPicFlag | | u(v) |
| | BOTTOM_FIELD_FLAG: | BottomFieldFlag | | u(v) |
| | IDR_PIC_ID: | IdrFlag | | ue(v) |
| | PIC_ORDER_CNT_LSB: | PicOrderCntLsb | | u(v) |
| | DELTA_PIC_ORDER_CNT_BOTTOM: | DeltaPicOrderCntBotom | | se(v) |
| | DELTA_PIC_ORDER_CNT_0: | DeltaPicOrderCnt0 | | se(v) |
| | DELTA_PIC_ORDER_CNT_1: | DeltaPicOrderCnt1 | | se(v) |
| | REDUNDANT_PIC_CNT: | RedundantPicCnt | | ue(v) |
| | DIRECT_SPECIAL_MV_PRED_FLAG: | DirectFlag | | u(v) |
| | NUM_REF_IDX_ACTIVE_OVERRIDE_FLAG: | OverrideFlag | | u(v) |
| | NUM_REF_IDX_10_ACTIVE_MINUS1: | NumRefIdx01Active_1 | | ue(v) |
| | NUM_REF_IDX_11_ACTIVE_MINUS1: | NumRefIdx11Active_1 | | ue(v) |
| | CABAC_INIT_IDC: | CabacInitIdc (aka ModelNumber) | | ue(v) |
| | SLICE_QP_DELTA: | SliceQpDelata | | se(v) |
| | SP_FOR_SWITCH_FLAG: | SpForSwitchFlag | | u(v) |
| | SLICE_QS_DELTA: | SliceQsDelata | | ue(v) |
| | DISABLE_DEBLOCKING_FILTER_IDC: | DisableDeblockFilterIdc (aka LoopFilterDisIDC) | | ue(v) |
| | SLICE_ALPHA_C0_OFFSET_DIV2: | LoopFilterAlphaOffset | | se(v) |
| | SLICE_BETA_OFFSET_DIV2: | LoopFilterBetaOffset | | se(v) |
| | SLICE_GROUP_CHANGE_CYCLE: | SliceGroupChangeCycle | | u(v) |
| | REF_PIC_LIST_REORDERING_FLAG_10: | RefPicListReordFlag10 | | u(v) |
| | REF_PIC_LIST_REORDERING_FLAG_11: | RefPicListReordFlag11 | | u(v) |
| | REORDERING_OF_PIC_NUMS_IDC: | RemapOfPicNumsIdc | | ue(v) |
| | ABS_DIFF_PIC_NUM_MINUS1: | AbsDiffPicNum10/11-1 | | ue(v) |
| | LONG_TERM_PIC_IDX_L0_1 | LongTermPicIsxL01 | | ue(v) |
| | LUMA_LOG2_WEIGHT_DENOM: | LumaLogWeightDenom | | ue(v) |
| | CHROMA_LOG2_WEIGHT_DENOM: | ChroLogWeightDenom | | ue(v) |
| | LUMA_WEIGHT_10_FLAG: | LumaWeightFlag10 | | u(v) |
| | LUMA_WEIGHT_11_FLAG: | LumaWeightFlag11 | | u(v) |
| | LUMA_WEIGHT_10_11: | LumaWeight | | se(v) |
| | LUMA_OFFSET_10_11: | LumaOffset | | se(v) |
| | CHROMA_WEIGHT_10_FLAG: | ChromaWeightFlag10 | | u(v) |
| | CHROMA_WEIGHT_11_FLAG: | ChromaWeightFlag10 | | u(v) |
| | CHROMA_WEIGHT_10_11: | ChromaWeight | | se(v) |
| | CHROMA_OFFSET_10_11: | ChromaOffset | | se(v) |
| | NO_OUTPUT_OF_PRIOR_PICS_FLAG: | NoOutOfPriorPicsFlag | | u(v) |
| | LONG_TERM_REFERENCE_FLAG: | LongTermRefFlag | | u(v) |
| | ADAPTIVE_REF_PIC_MARKING_MODE_FLAG: | AdaptiveRefPicBufering | | u(v) |
| | MEMORY_MGMENT_CONTROL_OPERATION: | MemMgmtControlOp | | ue(v) |
| | DIFFERENCE_OF_PIC_NUMS_MINUS1: | DifferenceOfPicNums-1 | | ue(v) |
| | LONG_TERM_PIC_NUM: | LongTermPicIdx10/11 | | ue(v) |
| | LONG_TERM_FRAME_IDX: | LongTermFrameIndex | | ue(v) |
| | MAX_LONG_TERM_FRAME_IDX_PLUS1: | MaxLongTermFrameIdx | | ue(v) |

**Appendix B: Retrieval of a context entry from ContextModeler depending on syntax type/sub-type. Also showing dependency of the handler on other data (to be provided through IMacroblock)**.

| | Dependencies (IMacroblock interface) | ContextIndex/ContextIncrement (relative to group base) |
|---|---|---|
| MB_TYPE: | Availability & mb_type of up/left neighboring MBs;<br>Slice_type (I/B/rest); | 1-13 bins; 3 unstructured code trees;<br>Integrated bin coding/context modeling;<br>Multiple calls to GetContextElement_Unstructured return both bin and context. |
| MB_SKIP_FLAG: | Availability & skip_flag of up/left neighboring MBs;<br>Slice_type (B/non-B); MB-type; Cbp; | 1 bin only, not BinIdx dependent;<br>CxtIncr = { 0, 1, 2} or {7, 8, 9} |
| SUB_MB_TYPE: | Slice_type (B/non-B); | 1-5 bins; 2 unstructured code trees;<br>Integrated bin coding/context modeling;<br>Multiple calls to GetContextElement_Unstructured return both bin and context. |
| MVD: Includes MVD_HORIZ, MVD_VERT context entries | Availability, mb_field of MBs containing neighboring sub-macroblocks;<br>MVD and position of neighbor sub-macroblocks;<br>Slice field/frame; | 1 bin + UEGK(3) of value1-1 + bypass coding of sign<br>CxtIncr = { 0, 1, 2, 3, 4, 5, 6 } |
| REF_PIC_INDEX: | Availability, mb_field, sub-block mode of MBs containing neighbor sub-MBs<br>Position of neighbor sub-MB;<br>Slice_type (B/non-B); Slice field/frame; | Generates 1 or more bins (dep. on unary coding);<br>bin 1 and 2: CxtIncr = { 0, 1, 2, 3}<br>other bins: CxtIncr = { 4, 5} |
| DELTA_QUANT_PARAM: | Last_dquant; | 1 bin + unary coding of value1-1;<br>Generates 1 or more bins (dep. on unary coding);<br>bin 1 and 2: CxtIncr = { 0, 1}<br>other bins: CxtIncr = { 2, 3} |
| INTRA_CHROMA_PRED_MODE: | Availability and c_ipred_mode of up/left neighboring MBs; | Generates 1–more bins (dep. on TU(2) coding);<br>bin 1: CxtIncr = { 0, 1, 2 }<br>other bins: CxtIncr = { 3 } |
| INTRA_PRED_MODE: Includes PREV_INTRA_PRED_MODE_FLAG, REM_INTRA_PRED_MODE context entries | | 1-4 bins;<br>bin 1 : CxtIncr = 0<br>bins 2-4: CxtIncr = 1 |
| MB_FIELD_CODING_FLAG: | Availability and mb_field of A/B neighboring MBs; | 1 bin only, not BinIdx dependent;<br>CxtIncr = { 0, 1, 2} |
| CODED_BLOCK_PATTERN:<br>CBPLuma = cbp % 16<br>CBPChroma = cbp / 16 | Availability, mb_type and cbp of upper/left MBs;<br>Availability, mb_type, position and cbp of MB containing left | 4 bins: 1 bin per each 4 luma blocks of cbp;<br>1 bin showing (chromaCbp !=0)<br>If true, 1 bin showing (chromaCbp ==2); |

| | | |
|---|---|---|
| | neighboring sub-macroblock;<br>Position of neighboring sub-macroblocks;<br>Slice_type (B/non-B);<br>Slice field/frame; | 5-6 bins;<br>bins 1-4: CxtIncr = { 0, 1, 2, 3}<br>bins 5: CxtIncr = { 4, 5, 6, 7}<br>bins 6: CxtIncr = { 8, 9, 10, 11} |
| CODED_COEFF_BLOCK:<br>Includes:<br>CODED_BLOCK_FLAG,<br>SIGNIFICANT_COEFF_FLAG,<br>LAST_SIGNIFICANT_COEFF_FLAG<br>COEFF_LEVEL | Availability, mb_type and position of MB containing upper/left neighboring sub-macroblocks; | 1) cbp_flag (1 bin)<br>- Its CxtIncr = [0, 19] (20 possible: 5 category, 4 context possible for each)<br>2) significance bit & last significant bit (for each coefficient); 61 contexts in total= 15+14+15+3+14 contexts for categories 0-4 and based on BinIndex:<br>- for significance bin: CxtIncr = [0, 60]<br>- for last significance bin: CxtIncr = [0, 60]<br>3) coeff levels (traversed from end of the block); 49 contexts in total (5 for 1st bin,4 for every next bin)<br>- 1st bin to show if abs value is equal to 1 CxtIncr = [0,4]<br>- for next bins: CxtIncr = [5, 49]; 4 bins for each binIndex of UExpG(14) coding of absolute value of coefficient-1<br>- 1 bin for sign of coefficient using bypass coding (no context needed) |
| END_OF_SLICE_FLAG: | EndOfSlice | CxtIndex = 276 |
| SLICE_HEADER:<br>And its subtypes: | | Using BaseCoder coding; not using CABAC context-modeller |