Model-driven Video Decoding: An Application Domain for Model Transformations

Christian Schenk, Sonja Schimmler, and Uwe M. Borghoff

Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
{c.schenk,sonja.schimmler,uwe.borghoff}@unibw.de

Abstract. Modern serialization formats for digital video data are designed in a way that they allow for the combination of high compression rates, high quality as well as performant en- and decoding. As the capability for real-time decoding often is a requirement, concrete decoders are usually implemented in a way that they best fit a specific architecture and, thus, are not portable. In scenarios, where it is essential that the capability to decode existent video data can be retained, even if the underlying architecture is changed, portability is more important than performance. For such scenarios, we propose decoder specifications that serve as templates for concrete decoder implementations on different architectures. As a high level of abstraction guarantees system-independence, we use (meta)models and model transformations for that purpose. Consequently, every decoder specification is (partially) executable, which simplifies the development process. In this paper, we describe our concepts for model-driven video decoding and give an overview of a prototypical implementation.

Keywords: model driven engineering; models; model transformations; video decoding

1 Introduction

In our current work, we are focusing on the problem of preserving digital video content. One challenge is to ensure that today's data can be restored on future architectures. There are several approaches that address this problem for general content [1]. One widely used approach is to use standard file formats that are likely to be readable on future architectures (such as PDF/A for documents and JPEG 2000 for images). But, to our knowledge, there is no such standard format for digital video content that is suitable for long-term preservation. The existing formats that are widely used (e.g., H.264 [4]) are designed for another purpose: they combine high compression and high quality, and they allow for the development of efficient en- and decoders. As these formats involve the use of lossy compression techniques, it is impossible to transform an already encoded digital

video into another format without risking an additional loss of information. As this implies that we should not introduce a new standard format, we now focus on the question of how decoding capabilities can be retained. As common decoder implementations are normally system-specific and thus not portable, we propose the definition of human-readable and machine-processable decoder specifications that serve as a template for the development of suitable decoders on different systems. Our goal are specifications that are partially executable and that allow a potential developer to create code (possibly supported by tools) that serves as a basis for the implementation of a functional (but possibly inefficient) decoder prototype for any existing video compression standard. As the capability to decode a digital video can be retained this way, existing videos can be preserved without any (additional) loss of information.

In our previous work [10], we introduced the general idea of our approach. In this paper, we are focusing on details of a protypical implementation that serves as a proof of concept. We will show how we combined (meta)models, model transformations, R scripts [9] as well as a control program in order to specify essential parts of the decoding process for H.264 encoded videos, and we explain how these parts constitute the basis for the implementation of an H.264 decoder. As digital video data tend to require lots of memory, we will also introduce our approach as a potential application domain for testing MDE tools and techniques in combination with large models.

The remaining paper is structured as follows: in section 2, we give a brief overview of video coding basics that are needed for the following sections. In section 3, we give an overview of our approach and describe some aspects in more detail. In section 4, we give an overview of related work before we conclude our work and give an outlook in section 5.

2 Serialization Formats for Digital Video Data

Usually, every (digital) video consists of *frames*, which, when presented consecutively for a fixed period of time, create the feeling of movement. Each frame mainly contains the image data and some time information (called the *composition time*) enabling a video player to play the video correctly. Due to the needed memory, storing a complete video using well-known image file serialization formats (such as png, jpg or bmp) is no option. Instead, in modern video compression formats, such as the H.264 standard [4], different compression techniques are combined in order to allow for suitable file sizes. Lossy compression algorithms play an essential role. Simply said, these algorithms do not distinguish between "similar" values; when stored, they are just handled equally. Consequently, these algorithms are irreversible, i.e., video encoding generally implies a loss of information. The H.264 standard, which is widely used and which will serve as a running example in this paper, is briefly introduced in the following.

2.1 H.264 basics

The H.264 standard is used for video streaming scenarios, in combination with Blue-Ray disks and, of course, also for locally stored video files. As it can be regarded as a de facto standard, we focus on H.264 encoded videos for our considerations. As other standards are specified similarly if not identically (e.g., MPEG-4 AVC), we assume, however, that it is straightforward to use our approach for these standards.

An H.264 video track is structured in *samples*, which are (usually) grouped into *chunks*. Each sample can be seen as a (still encoded) representation of a single frame in the video, i.e., it contains every information that is needed to reconstruct a two-dimensional field of pixels. Pixel data are usually stored using the YCbCr color model, which distinguishes between one luma (Y) and two chroma channels (Cb and Cr), whereby each channel is encoded independently.

Compression basics: Principally, each sample is encoded using lossy image compression. However, in addition to that, *delta compression* is used in order to remove redundancies caused by pixel similarities between successive frames and (spatially) nearby regions within one frame. We briefly explain how it works: in case of a single frame, it is common that pixels are similar or even equal if compared with neighboring pixels. Furthermore, two successive frames often only differ in details (in order to allow for smooth frame changes). The delta compression's main principle is to just store "quantified" difference values between two similar pixels. The quantification, which is a simple integer division, just increases the desired effect: the resulting values tend to be smaller and occur more often. (By the way, the fact that the integer division is not completely reversible is one of the reasons why the complete compression is lossy.)

In an H.264 encoded video, every sample is divided into a grid of 16 x 16 blocks (called macroblocks), which are traversed row by row and column by column during the decoding process. Each macroblock may depend on one or more other macroblocks whose data already have been decoded before. When a macroblock is decoded, the data of its dependencies are used to first *predict* [4] an intermediate data representation. Afterwards, the explicitly stored difference values are added in order to restore the macroblock's actual data. Macroblocks that only depend on data of neighboring macroblocks (within the same frame) are called *intra-predicted*, whereas most of the macroblocks of an H.264 encoded video are usually *inter-predicted*, i.e., they refer to arbitrary regions of other frames using *motion vectors* (in combination with frame ids).

Inter-predicted macroblocks automatically cause inter-dependencies between different frames. In order to constrain possible inter-dependencies (which is essential when a video is not decoded from the beginning), the sequence of all frames are partitioned into groups of pictures (called GOPs), whereby two frames can only depend on each other if they belong to the same GOP. Consequently, every GOP contains one or more (intra-predicted) frames that do not depend on any other frame and thus only consists of intra-predicted macroblocks. All the

other (inter-predicted) frames contain inter-predicted macroblocks and depend on one or more other frames.

Even if the delta compression plays an essential role for the complete encoding, it is more a preparation for the actual compression: the results of the delta compression are compressed using variable length encoding (e.g., huffman encoding), which ensures that more frequent values are transformed into smaller codes than values that are less frequent.

Decoding order and composition order: An H.264 encoded video permits sequential decoding, i.e., forward jumps are generally not necessary during the decoding process. A frame can only be restored if all the frames it depends on have already been decoded. Hence, if a frame A depends on a frame B, B must be stored before A within the serialization. Nevertheless, the H.264 standard allows frames to depend on other frames that have a greater composition time, i.e., which have to be displayed later during the playback. That is why we have to distinguish between the *decoding order* and the *composition order*: frames are serialized in decoding order but have to be presented in composition order. Thus, after their restoring in decoding order, all the frames have to be put into composition order.

3 Model-driven Video Decoding

In this section, we will give an overview of our approach of model-driven video decoding. We will further give some details of the model-based parts.

3.1 Overview of the Approach

Regardless of the actual scenario and the further processing, the main task of any decoder is to transform an encoded binary representation (e.g., an H.264 serialization) into a sequence of frames. Therefore, we have decided to use a suitable abstraction of that principle as a basis for our approach.

As illustrated in Fig. 1, the model-driven decoding process is divided into 5 phases: The *modeling phase* converts the original video into a model representation that corresponds to the H.264 metamodel, which formalizes H.264 content (see Fig. 2). The *preparation phase*'s purpose is to partition the complete work into independent "parts", which we call *task chains* in the following. Each task chain contains all the information that is needed to decode one GOP. In the *execution phase*, every task chain is actually decoded, i.e., the dependencies are resolved, and the pixel data are restored. In the *finalization phase*, the result of the execution phase is sorted in accordance to the composition order and transformed into a model that corresponds to the *frame sequence metamodel*, which formalizes general video content (see Fig. 3). The *unmodeling phase*'s purpose is simple: it transforms such a model into the final result, i.e., into a sequence of concrete frames.

In order to describe the decoding process in an architecture-independent way, our decoder specifications unify different abstraction mechanisms. As the

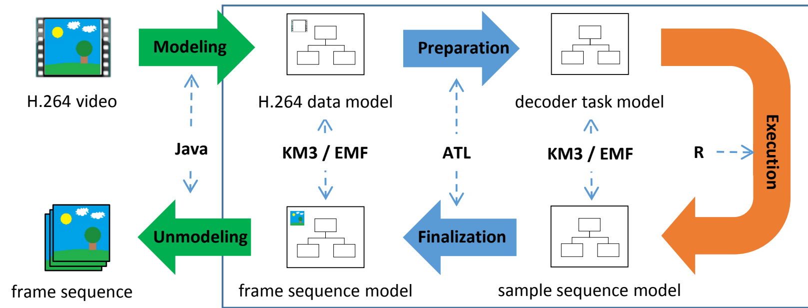

Fig. 1. Model-driven video decoding

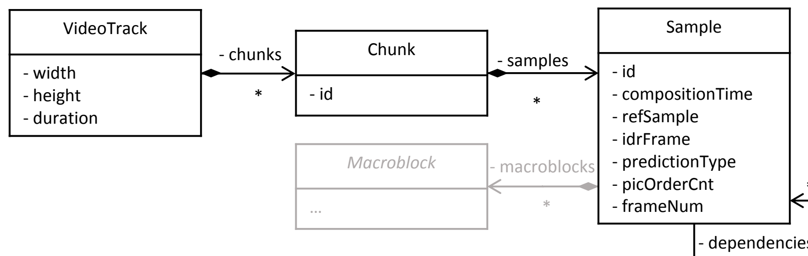

Fig. 2. H.264 input metamodel - due to clarity reasons, details of the abstract class `Macroblock` (colored in gray) and all its subclasses have been omitted.

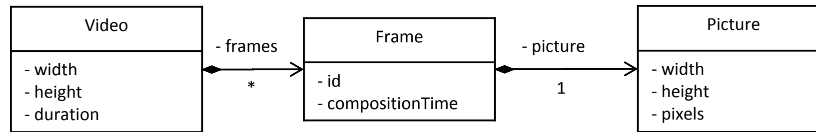

Fig. 3. Frame sequence metamodel

modeling and the unmodeling phase essentially perform pre- and post-processing steps, which depend on the underlying architecture and on the concrete scenario, they are not in the scope of such a specification.

In the next section, we give some details of how the three remaining phases have actually been implemented.

3.2 Details of the Approach

For the formalization of all the intermediate data representations, which serve as in- and output for the different phases, we use EMF-based [11] metamodels. Consequently, the complete decoding process may be regarded as a series of model transformations. Indeed, we use model transformations as the means to

formalize the preparation and the finalization phase. Within the execution phase, however, the inter- and intra-predicted values must be determined in order to restore the pixel values. As this process involves different mathematical calculations, we have decided to describe this phase using a mathematical abstraction. As we want our decoder specifications to be human-comprehensible as well as machine-processable, we have decided to use languages that are text-based. In summary, we use ATL [5] model transformations, KM3 [6] defined metamodels as well as R scripts [9]. We will give some details in the following:

KM3-based metamodels: Being convenient to define EMF metamodels using simple text, the KM3 language has been used to define all the necessary metamodels.

For the preparation phase, for example, we have defined 5 metamodels, which are used for the formalization of 7 (internal) models. The following excerpt shows the definition of the class `Frame` of the frame sequence metamodel (see Fig. 3).

```
class Frame {
  attribute frameId : Int32;
  attribute compositionTime : Int64;
  reference picture container : Picture;
}
```

ATL model transformations: The transformations of the preparation and the finalization phase are written in ATL, using mostly its declarative language features. We have decided to use ATL because it provides declarative but also imperative features, is based on EMF, is well integrated into the Eclipse IDE and can also easily be executed programmatically. Generally, every language that fulfills these requirements is an appropriate alternative for ATL (such as the Epsilon Transformation Language (ETL) [8]).

For the preparation phase, for instance, we have defined 5 ATL transformations, containing 16 rule definitions (with a total of about 190 LOCs). One example is a transformation that creates a model that explicitly stores the GOPs. The following excerpt shows the responsible ATL rule:

```
rule VideoTrack2Video {
  from
    vt : H264!VideoTrack
  to
    v : GOP!Video (
      gops <- vt.gopSmpls->collect(smpls | thisModule.createGOP(smpls))
    )
}
```

The ATL *helper* `gopSmpls` does the actual partitioning and determines a sequence of sample sequences representing the GOPs.

R scripts: For the abstraction of the mathematical operations we use R scripts. The R language was originally designed for statistical calculations. We have chosen to use it for our approach as it allows us to express and execute all

the mathematical operations that we need for the decoding (including matrix operations). Principally, every language that provides basic as well as matrix operations, e.g., Octave[1], would be a suitable alternative.

For the determination of the luma and chroma values, for instance, we have defined 25 R scripts (with a total of about 600 LOCs). The following excerpt, for example, shows how the luma values of a macroblock are determined. The variable `prediction` contains the intra-predicted values, whereas the parameter `residual` contains the (pre-processed) delta values. (The function `Clip1Y` simply ensures that the result is in the correct range).

```
for (x in 1:16) {
  for (y in 1:16) {
    values[x, y] <- Clip1Y(residual[x, y] + prediction[x, y])
  }
}
```

DSL-based control program: For coordination, we use control programs written in a DSL that we have defined with the framework Xtext [2]. Xtext is a framework that allows for the definition of DSLs (and the generation of corresponding tools) based on a grammar specification. Such a control program constitutes the frame of the specification as it defines how (meta)models, transformations and R scripts are connected. The following example shows how metamodels and models are declared within the the control program:

```
metamodel MM_H264 origin h264.km3.ecore
model H264 conforms to MM_H264 origin in.h264
```

The next excerpt shows how a transformation `H264ToGOP` is introduced, which is stored in `H264ToGOP.asm` and transforms the model `H264` into a model `GOP`.

```
transformation H264ToGOP origin H264ToGOP.asm {
  input model H264
  output model GOP
}
```

3.3 Implementation Status

As explained before, essential steps of the decoding work have already been specified using different abstraction mechanisms. Each step we have abstracted so far was originally implemented in form of a Java tool set; consequently, we have a reference implementation permitting us to generate suitable test data for every abstracted decoding step. We have developed a Java library that allows us to access and extract all information of an H.264 encoded video. This library is used for the implementation of the modeling phase. Furthermore, we have developed a Java application that is able to decode every intra-decoded frame

[1] Project URL: www.gnu.org/software/octave

of an H.264 encoded video file (stored in the mp4 file format). We have also developed a Java tool that can process control program files.

Up to now, the specification of the preparation phase is complete. In combination with the control program definition, an H.264 model can automatically be transformed into a decoder task model (containing the task chains).

We have also already defined all the necessary R scripts that are needed to decode intra-predicted macroblocks, and we have tested them with our Java application, which uses the open source library renjin[2] for interpreting the R code. In a next step, we want to integrate the execution phase into the decoder specification. The missing link is a conversion of the model-based representation into an R-compatible input format. Here, we want to check if this conversion can be implemented using languages that allow for the specification of model-to-text transformations (such as the Epsilon Generation Language (EGL), which is related to ETL [8]).

The finalization phase has also already been specified, but will certainly need an update when the execution phase is integrated.

3.4 Scalability

We have chosen abstraction mechanisms that are executable in order to simplify the development process of functional (but not necessarily efficient) decoders for arbitrary architectures. Supposing that the most effective optimization measures are system-specific and therefore decrease the level of abstraction, we generally have neglected any performance considerations within the design of the specifications.

For practical reasons, we have made one exception: all the models within the preparation phase do not contain any macroblock data as they are not needed before the execution phase. Consequently, any H.264 model (that conforms to the metamodel illustrated in Fig. 2) does not contain instances of the class `Macroblock` (and its subclasses). As these instances contain (i.e., within referenced objects) the information to restore the actual pixel data, the size of H.264 models can drastically be reduced.

For an evaluation, we generated the H.264 models for three different videos and determined their sizes before and after removing the macroblock data. As models can be regarded as graphs, we simply counted the number of nodes and edges to specify the models' size. Furthermore, we measured the time it took to execute the preparation phase and determined the percentage of this time spent on the model transformations (MTs). Finally, we measured the time it took to decode all the intra-predicted frames (I-frames) when using the renjin-based Java application. The results are listed in Table 1.[3]

Currently, the frames are decoded sequentially. But as a GOP can completely be decoded independently of other ones, decoding them simultaneously is assumed to crucially decrease the execution time of the decoding process.

---

[2] Project URL: www.renjin.org

[3] Used abbreviations: k for kilo ($\times 10^3$), M for mega ($\times 10^6$)

Table 1. Evaluation results for test videos (System: Debian 8, CPU: i7-4790 3.6 GHz, RAM: 32 GB)

	video 1	**video 2**	**video 3**
Frames (I-frames)	2540 (52)	8189 (273)	163327 (2373)
Resolution (w × h)	512 × 288	1920 × 1080	1280 × 720
Model graph's nodes/edges			
- original	21.6M/24.6M	1760M/1900M	14500M/15700M
- reduced	5.08k/13.0k	9.04k/24.6k	327k/646k
Preparation phase (MTs)	∼ 2 s (75%)	∼ 5 s (83%)	∼ 7 min (96%)
R-based decoding of I-frames	∼ 24 min	∼ 1 day	∼ 4 days

4 Related Work

Our approach utilizes MDE techniques in the domain of long-term preservation in order to address the issue of preserving digital video content in a way that it can be restored on future architectures. As far as we know, there are no similar approaches. In the following, we present two approaches based on image encoding standards.

Motion JPEG 2000 [3] is a part of the JPEG 2000 standard that simply uses the corresponding compression for each frame. As JPEG 2000 is already used for the preservation of digital images [7], a combination would principally allow for the preservation of digital video content. In another approach [12], digital videos are preserved using an XML-representation. Two variants are distinguished: first, video data are completely transformed into primitive XML. Second, the video frames are converted into image files that are suitable for long-term preservation (such as JPEG 2000) and referenced within the XML representation. Both variants involve the decompression of lossy compressed parts, which naturally leads to larger file sizes.

An important difference between our approach and the approaches named before is that we generally allow for delta compression, i.e., frames can have dependencies between each other. Avoiding delta compression simplifies the decoding, but results in larger file sizes. Contrary to our approach, the two approaches also imply recoding, which results in potential loss of information.

5 Conclusion and Outlook

In this paper, we have continued our presentation of a model-based approach for architecture-independent video decoding [10] with a focus on the model-based parts and its realization. At this point, our approach does not provide full support for the decoding of inter-predicted frames, and the conversion between models and R-compatible representations is still hard-coded in Java. Nevertheless, so far we have succeeded to find appropriate abstractions for all essential parts of the decoding process.

Our approach involves the processing of large models. Even if the efficiency aspect plays a subordinate role, optimizations, which do not impede the applicability, might improve the development process. Besides, the utilized models may also be used as potential test data for common MDE tools.

After finishing the specification of the H.264 decoding process, we plan to focus on the design of a serialization format that simplifies the implementation of the modeling phase, can be used to preserve H.264 encoded data without additional loss of authenticity and allows for “suitable” file sizes.

In a further step, we want to evaluate our approach by asking developers who are unfamiliar with the H.264 standard to implement a decoder based on our specification. In this context, we also want to find out whether and in which way our decoder specifications can also be used to simplify the development process by serving as a base for automatic code generation.

References

1. Borghoff, U.M., Rödig, P., Scheffczyk, J., Schmitz, L.: Long-Term Preservation of Digital Documents, Principles and Practices. Springer-Verlag Berlin Heidelberg (2006)
2. Efftinge, S., Völter, M.: oAW xText: A Framework for Textual DSLs. In: Eclipsecon Summit Europe 2006 (2006)
3. ISO/IEC: International Standard ISO/IEC 15444-3: JPEG 2000 Image Coding System - Part 3: Motion JPEG 2000. International Standard Organization (2002)
4. ITU-T: Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services. International Telecommunication Unit (2013)
5. Jouault, F., Allilaire, F., Bzivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(12), 31–39 (2008)
6. Jouault, F., Bzivin, J., Team, A.: KM3: A DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems. pp. 171–185. Springer (2006)
7. van der Knijff, J.: JPEG 2000 for Long-term Preservation: JP2 as a Preservation Format. D-Lib Magazine 17(5/6) (2011)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: International Conference on Theory and Practice of Model Transformations. pp. 46–60. Springer-Verlag, Berlin, Heidelberg (2008)
9. Ross Ihaka, R.G.: R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics 5(3), 299–314 (1996)
10. Schenk, C., Maier, S., Borghoff, U.M.: A Model-based Approach for Architecture-independent Video Decoding. In: 2015 International Conference on Collaboration Technologies and Systems. pp. 407–414 (2015)
11. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
12. Uherek, A., Maier, S., Borghoff, U.M.: An Approach for Long-term Preservation of Digital Videos based on the Extensible MPEG-4 Textual Format. In: 2014 International Conference on Collaboration Technologies and Systems. pp. 324–329 (2014)