Using Java to prototype a H.264/AVC decoder

Marek Parfieniuk [1], Alexander Petrovsky [1], Alexey Petrovsky [2]
1) Department of Real-Time Systems, Bialystok Technical University,
Wiejska 45A, 15-351 Bialystok, Poland,
marekpk@wi.pb.edu.pl, palex@bsuir.by, http://aragorn.pb.bialystok.pl/~marekpk
2) Computer Science Department, Belarusian State University of Informatics and Radioelectronics,
Brovki 6, 220013, Minsk, Belarus,
petrovsky@bsuir.by, http://www.bsuir.by

Abstract: *This paper presents our experiences in using Java to prototype a H.264 decoder and to develop accompanying tools: GUI-based diagnostic applications and demos of subalgorithms. The project is aimed at preparing a reliable basis for implementing the video coding standard in hardware. The pros and cons of the Java programming language are discussed in terms of developing both such advanced DSP algorithms and applications using them. Especially, high productivity is pointed out as an advantage of Java over C/C++, which is related not only to the language itself but also to the rich toolset provided as the bundle of the JDK and NetBeans.*

Keywords: H.264, decoder, Java, implementation.

1. INTRODUCTION

Recommended by both ITU-T and ISO/IEC [1-6], H.264/AVC (Advanced Video Coding: MPEG-4 Part 10) is the most recent and innovative standard of video coding, which has widespread industry adoption as a foundation of new multimedia standards, services, and products. Especially, it has been chosen to be used for Blu-ray Disc, digital broadcasting via DVB, 3GPP mobile communication, teleconferencing, and media streaming over the Internet.

This huge application area is a consequence of the great flexibility of H.264, which was designed to be network-friendly [2]. The encoder, which is equipped with a lot of options and parameters, allows for customizing its output bitstream in order to best fit a given application. Especially, the stream can be adjusted to the display resolution and computational power of end-user terminals, and bandwidth usage can be traded off for the visual quality.

Another advantage of the codec over its predecessors is much lower usage of bandwidth, as H.264 offers compression efficiency even two times greater than that of MPEG-2 Video (H.262), at comparable quality of the reconstructed video. This has been achieved by incorporating into the algorithm the most notable and recent techniques of video coding, which, however, require increased computational load [7, 8]. In particular, decoding can require even 4 times more operations than that of MPEG-2.

In recent years, there is a significant effort to build infrastructure that supports H.264. Real-time encoding engines for broadcasting purposes, HDTV-sets, next-generation media players, and mobile devices need coprocessors that speed up particular stages of the algorithm or, even better, chips that realize the entire encoding or decoding task.

Implementing H.264 is not trivial for several reasons. As the standard is not a simple extension of the previous ones, only very limited reuse of existing hardware and software is possible. From another point of view, the algorithm complexity makes design difficult, especially if a small and energy efficient device is expected to operate in real time. Finally, the standard still evolves, so that code organization or chip architecture must be made flexible, which allows it to be easily adopted to handle future extensions of H.264 or to maximize performance in a particular application.

The authors have undertaken the challenging task of addressing these issues. They work on developing a new flexible and optimized architecture of a H.264 decoder that is computationally efficient and allows speed to be traded off for resource consumption. The decoder additionally has to be modularized and reconfigurable, so that functionalities of new standard revisions can easily be added without redesigning the whole system. We expect at least simplified development of new circuits and easy customization of a chip in order to best match application requirements.

In order to complete the task, the team has decided to design a new object-oriented model of the decoder from scratch and implement it in software. The developed classes, after testing their source code, will define hardware modules, whose functional verification can be based on data generated using the software.

An accompanying decision was to use the Java platform in object-oriented development. Applying this advanced technology was expected to increase productivity, to make results more reliable and reusable, and to reduce investments. The issue of real-time performance of prototype programs was pushed into the background, as unnecessary in such circumstances.

The paper presents our motivations for using Java, issues related to this decision, and the appealing results obtained. It is shown that a methodology based on the high-level language, with is supported by a rich toolset, allows for quick prototyping advanced DSP algorithms like H.264 and for producing well-documented, self-describing code which can serve as a basis for implementing real-time hardware. Moreover, an extensive easy-to-use GUI-based environment for verification and testing can easily be developed in parallel.

2. H.264 STANDARD

The H.264 video coding standard, also called the MPEG-4 Advanced Video Coding (AVC), has been finalized in March 2003. Its development began in 1997 with the aim of achieving better coding performance

compared to up-to-data standards, particularly MPEG-2, and greater flexibility from the point of view of network applications. The Joint Video Team (JVT), a group of experts of both ITU and ISO, has then been formed, which is responsible for developing and maintaining the standard.

Fig.1 shows the general scheme of a H.264 codec, which, like the older standards, is a hybrid algorithm that removes both spatial and temporal redundancy of video signals by combining transform coding with motion-compensated predictive coding. The main principle has been left unchanged because it was possible to achieve better flexibility and compression efficiency by only improving subalgorithms.

Motion is estimated more precisely, and thus compensated more effectively, with quarter-pixel accuracy and fine-grained partitioning of macroblocks into smaller units. Moreover, an in-loop deblocking filter removes the blocking artifact before using a frame for prediction, which further improves estimation accuracy. Temporal redundancy is also better removed, because it is possible to use multiple (up to 16) reference frames. The bidirectional prediction allows future frames to be referenced in addition to past ones.

Removing spatial dependencies among pixels by a decorrelating transform can be supported with multi-mode intra prediction of a block using adjacent fragments of the same frame. Moreover, transform size can be switched between 4×4 and 8×8 in order to best fit macroblock contents.

Finally, more effective methods of entropy coding have been employed: Context-based Adaptive Binary Arithmetic Coding (CABAC) and Context-Adaptive Variable Length Coding (CAVLC).

All these techniques improve coding performance at the price of increasing computational demands. The complexity of H.264 is estimated to be 5-8 times that of H.263, even though the standard uses efficient multiplierless transforms to approximate the Discrete Cosine Transform (DCT).

H.264 specifies several profiles, which address various applications i.e. different trade-offs among quality, bitrate, and computational requirements. The Baseline, Main, and Extended profiles are of primary importance. The first one has modest computational demands at the price of quality, the second takes full advantage of the coding algorithm, whereas the third is best suited to streaming applications. Recently, the standard has been extended toward higher fidelity (sample bit depth greater than 8 bits) and scalable coding, as Fidelity Range Extension (FREXt) and Scalable Video Coding (SVC) have been added [3, 4].

It is important that H.264 uses patented techniques, and thus including an implementation of some of its profiles into a commercial product requires paying royalties to patent holders.

3. EXISTING SUPPORT FOR IMPLEMENTING H.264

H.264 is described in a huge standard document [1], whose several versions exist. This of 3/2005 amounts 343 pages, whereas that of 11/2007 consists of 564 pages, which is mainly because of adding the SVC extension.

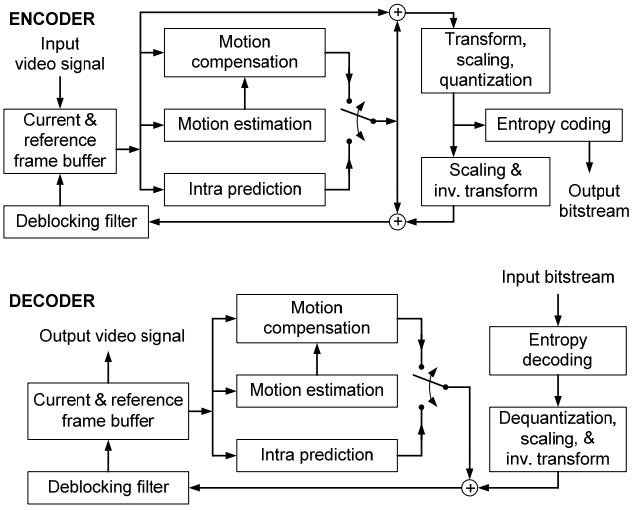

Fig.1 – General scheme of the H.264 codec.

Especially for non-experts in the field of video coding, the document is very difficult to read and interpret, because it is not only complex but also written in a very peculiar manner. All information necessary to create interoperable software or hardware is provided, but there are no suggestions how to implement it. Especially, there is no word about the encoding process: only the bitstream format and decoding process are specified using text, tables and C-like pseudocode.

The document is hardly useful even for implementing a H.264 decoder, because of far-reaching formalism, widely-used cross-referencing, and the complexity of the standard itself, which is manifested by a huge number of options, definitions, and variables. Even the pseudocode cannot be directly used as a basis for a solution, as it describes only bitstream parsing, while decoding is presented by means of text. Most of variables are not clearly defined, so that their meanings and data types must usually be deduced from several fragments of text.

As to document versions, it should be noted that apart from adding extensions and correcting bugs, changes are made in the whole text. In practice, this makes it impossible to easily switch from one publication to another. Moreover, some modifications are not well documented, e.g. in the version 11/2007, the residual colour transform has been removed without explicit notification.

Careful following even subtle changes of the standard is a must that makes development even more difficult. In order to be up to date, designers must observe activities of the JVT (which makes publicly available some draft documents and reports on meetings) or, even better, to participate in their works. Either approach is acceptable only for big companies.

In order to help developers, the standard is supported with a reference software [9, 10], which is developed by the JVT. It is called the Joint Model (JM) and consists of both encoder and decoder written in the C language. They allow custom standard implementations to be validated, provide encoding statistics and video quality assessment, but are not very helpful in developing hardware or software.

The huge piece of code (over 50000 lines) is neither well documented nor organized. Only a user guide is provided, not thorough design documentation. Comments are placed sparingly, and abbreviated identifiers are unclear. Pointers, type casting, and global variables are widely used, which makes debugging difficult. The code is also far from a clear object-oriented design, even if it is modularized by grouping related routines into separate files. It is also not optimized for performance.

At the time of writing these words, the version 15.1 of the JM is accessible. Frequent updates are limited to adding extensions and fixing bugs. The code seems not to be refactored in order to improve its quality. In [5], it is explained that this is a consequence of the politics the JVT selected to develop the standard. Every proponent of an option or extension, after proving its usefulness and then finding acceptance, must integrate it with the existing reference software. Obviously, he has no interest in improving someone else's code, the more so because the result will be available to everybody for free, whereas a good implementation of H.264 still has commercial value for many companies.

Because of all of these, understanding the reference software and taking its advantage are very difficult even for experienced programmers. It is even problematic to extract data from key points of processing pipeline, which is necessary for verification purposes. Detailed tracing of bitstream contents is supported, but accessing data of the algorithm core requires code modification, which is not easy for the reasons mentioned above.

It should be emphasized that our opinion about the reference code conforms those of other developers, which are reported over the Internet [11, 12].

Internet forums and mail-lists, like [11] or [12] are good sources of interesting information for a developer. They are active, moderated by practitioners, and contain a lot of knowledge in the form of brief messages, free of embellishments. Others' experiences and advices are very useful in understanding standard nuances, explaining doubts, solving problems and planning development.

Another noteworthy fact is that there exist initiatives to develop open-source H.264 software: x264 [13] and libavcodec [14]. Even though they outperform the JM in terms of performance, it is also hard to consider them as a good and reliable basis for developing own hardware solutions. Firstly, they as well as the JM lack both object-oriented design and in-depth documentation. Secondly, they are not very credible in terms of both standard conformance and development life-cycle: some options can be omitted in order to simplify design, and similar initiatives often became inactive even before reaching a mature state.

The journal papers [1-5] and book [6] about H.264 seem the best basis for beginning high-level design of a decoder. They describe main principles, which is sufficient to identify main classes and methods. Neither the standard document nor reference software are helpful in this regard, but they are an invaluable source of technical details and nuances when switching to coding. The document is more useful in implementing subalgorithms, whereas the reference software allows for testing them and explaining doubts.

As to testing, there are some analyzers of H.264 streams, like H264Visa [15], but from our point of view they have clear disadvantages. First of all, they offer only limited access to interesting points of the decoding pipeline. Some data can be viewed via GUI, but there is no mechanism to automatically translate them into a form suitable for verification. As such programs are provided without source code, it is impossible to extend them to satisfy our needs. Additionally, they come at significant prices and require Microsoft Windows, so that cannot be run on Linux.

Putting together the above facts, we can conclude that existing support for implementing H.264 is quite extensive but very inconsistent and difficult to use. There are no ready-to-use patterns, universal tools, and explicit design insights. In order to develop a really efficient hardware decoder in a reasonable time and without much investment, one must develop his own architecture, work methodology, and software tools.

Thus far our team worked mainly in the fields of speech coding and enhancement [16, 17], so that we had no much earlier experiences with video codecs. Developing dedicated software in parallel to hardware was a mean to gain more practical knowledge about video processing and to avoid bad decisions at hardware design.

4. JAVA AS A TOOL FOR IMPLEMENTING H.264

Implementing from-the-scratch such an advanced algorithm as H.264 is a challenging task. In order to focus only on architectural issues, it is important to avoid problems with development and coding, in which advanced tools are helpful [18, 19]. In particular, high productivity is mainly related to early detection of bugs or, even better, to preventing them from arising. Apart from desiring functionality, we expect tools to be accessible free-of-charge, well supported, and easy to use.

The conservative approach of using the C language is widely considered as of poor productivity. A lot of care is necessary to write a reliable code and make it portable, even between Windows and Linux. Memory management is left to a programmer, which distracts him from the algorithm. Due to limited type safety, many bugs are possible and usually difficult to detect, especially those related to exceeding array bounds, type casting, and pointers.

It is much better to use C++, which is a more advanced language that well supports object-oriented design. In spite of better type safety, errors of a wide range are still possible, as pointer-based memory access and management cannot be completely avoided.

Another approach is to generate code from a schematic model, which is possible e.g. with Matlab-Simulink. This completely prevents programming errors but simultaneously make it difficult to customize and extend the code obtained. Such tools are also expensive and usually do not support well exporting projects or their fragments to other development environments.

These facts have motivated us to develop H.264 software using Java: a language which recently focuses attention of developers of real-time and embedded systems [18-24].

Java has been developed in the mid-1990s by Sun Microsystems with the aim of facilitating platform-independent programming and improving productivity.

This modern object-oriented language has been equipped with a lot of practical features such as threads, assertions, built-in security, and automatic memory management. Moreover, the Java Development Kit (JDK) is free and comes with a huge set of libraries for different purposes: from dynamic data structures and advanced string manipulations to networking, Graphical User Interface (GUI), or even multimedia [25]. Even an advanced Rapid Application Development (RAD) environment, the NetBeans, is provided.

Of course there are equivalent libraries and tools for C/C++, but they are often costly and available as separate items, so that much effort is necessary to configure and maintain a developer workstation.

On the other hand, Java is more a technology than just a language, which is sufficiently powerful and universal to be useful in almost all applications, except only those in which performance and memory usage are critical. The latter is because Java portability has been achieved by making the language interpreted. Compilation results in hardware-independent byte code which is run on the Java Virtual Machine (JVM). This piece of software obviously represents some execution overhead, which is additionally unpredictably affected by garbage collection of automatic memory management.

Moreover, even though byte code is itself compact, even simplified versions of the JVM need hundredths kilobytes of memory, whereas taking advantage of rich libraries requires megabytes. This is often unacceptable in embedded applications.

In addition to these problems, the peculiarities of a target platform often make porting the JVM to it difficult or at least not economically justified. Even on PC, we have observed incompatibility issues. Strange errors sometimes occur if an application compiled for some version of the JVM is run using an older one.

In spite of difficulties, Java advantages sustain the interest in introducing Java to embedded and real-time systems [18-24]. For example, Sun's picoJava processor and ARM9J with the Jazelle coprocessor are examples of efforts to implement an efficient hardware-accelerated JVM. On the other hand, programming techniques are developed which allows for overcoming Java limitations [21, 22]. Finally, the technology is extended to satisfy specific needs of demanding applications.

Thus far, the main success of these efforts is the popularity of the Java 2 Micro Edition (J2ME), a tailored and thus lightweight programming platform for mobile phones.

It should be emphasized that the aim of the present work is to develop a software basis for implementing a hardware H.264 decoder. We do not use Java in an embedded system. Nevertheless, our results can obviously serve for the latter purpose, so that such their utilization is conceivable in the future, in the context of J2ME and Java Media Framework (JMF) [25].

5. PROJECT RESULTS

Our project has reached the half-way point. The object model of a H.264 decoder has been developed, as well as a preliminary design of the corresponding hardware architecture. Most of functionalities have prototype software implementations. Parts of the decoder whose code had reached a stable form and had been thoroughly tested, have been implemented in FPGA. Interconnections among functional units and memory as well as essential control logic have also been developed, which is the first step in assembling the final chip.

Especially, the parser, VLC decoding, and transform blocks are nearly finished. What is important is that good software prototypes allowed hardware engineers to quickly understand which digital circuits are expected and to construct them not only efficiently but also optimally, i.e. high performance has been achieved at low resource utilization. This will be described in future papers.

As to the software implementation, it consists of about 50 classes, which are shown in Fig. 2 as a Unified Modeling Language (UML) diagram, which also presents main relations between them. Most of classes can be directly identified with hardware modules. Carefully designed methods define control registers and state changes.

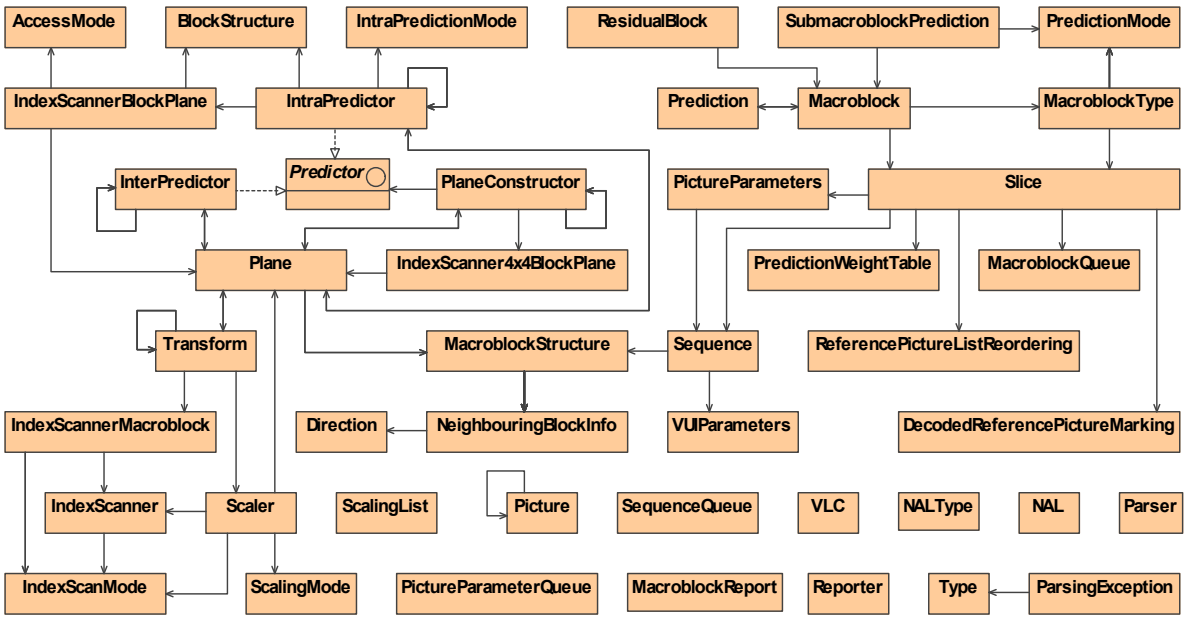

Fig.2 – UML class-diagram of the object model of a H.264 decoder.

Most of classes have strictly determined numbers of instances. Obviously, there is only one stream parser and one VLC decoding engine, whereas most of the remaining blocks of Fig.2 must be tripled in order to decode luma and both chroma components using separate pipelines, which can efficiently work in parallel.

Knowing the number of objects of a particular class, allows them to be preallocated as static fields and to exist continuously during program execution. This significantly reduces computational load related to memory management and garbage collection. It seems that using this technique is crucial for developing a Java-based H.264 decoder that works in real time.

Another conclusion, which does not directly result from the standard document, is that most of operations can be performed without explicit integer multiplications. The latter can widely be replaced with binary shifts, possibly supplemented with additions.

As to data types, 16 bits (including sign) seem sufficient to store variables related to decoding, but in some cases, auxiliary results need 32 bits. Most of data represented in the standard document using tables of integer numbers can be efficiently packed into compact bit strings, which conserves resources.

Internal variables of decoding pipelines take not so much memory. Quantization tables and sample buffers for transform and prediction purposes occupy most space, yet it seems possible to incorporate them into a chip. The main problem is in storing reference frames for inter-prediction, which requires large out-of-chip memory. Some of known decoders require encoders to limit the number of reference frames depending on video resolution and accessible storage space, and we will probably employ this approach in our chip.

High fidelity extensions of H.264, in which samples are represented using 10 or 12 bits, instead of the typical 8 bits, require specific memory architecture or wasting space.

A notable result of the project is a platform-independent diagnostic tool, which works in any operating system equipped in the JVM, especially on Linux. It reuses the code of the software decoder we have developed, so that the latter can be tested and demonstrated interactively via GUI. The tool consists of two modules, whose main windows are shown in Fig.3 and Fig.4. The former allows H.264 streams to be analyzed and restructured, in order to focus tests on fragments that cause the decoder to fail. The second module allows a single frame to be examined: decoding correctness can be verified both visually and by following dataflow step by step. The latter required a quite advanced reporting mechanism to be developed, which can be easily and consistently incorporated into decoder and collects data in a synthetic form, so that they can be both displayed on screen and exported to verification tools.

The reporting and verification functionalities still need to be enhanced. Especially, filters are to be developed that allow interesting information to be quickly extracted. Another lacking option is automatic detection of erroneously decoded frames in a long stream, and macroblocks in a picture. Nevertheless, interactive testing the programs support is sufficient in most cases.

Side-effects of our work are several applications that demonstrate subalgorithms of H.264 and explain data structures it uses. For example, Fig.5 shows the main window of the tool that allows users to interactively study Picture-Adaptive Frame/Field (PAFF) and Macroblock-Adaptive Frame/Field (MBAFF) modes of accessing image samples.

A final remark is that JavaDoc, a tool for generating well-organized HTML documentation from source code comments, whose effect is shown in Fig.6, has proved itself to be a very useful and effective communication means, which allowed software developers to impart their knowledge to designers of hardware modules, without producing many extra reports.

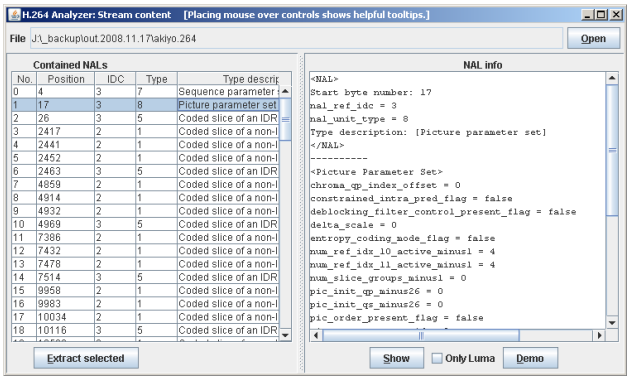

Fig.3 – GUI-based diagnostic tool: stream analysis.

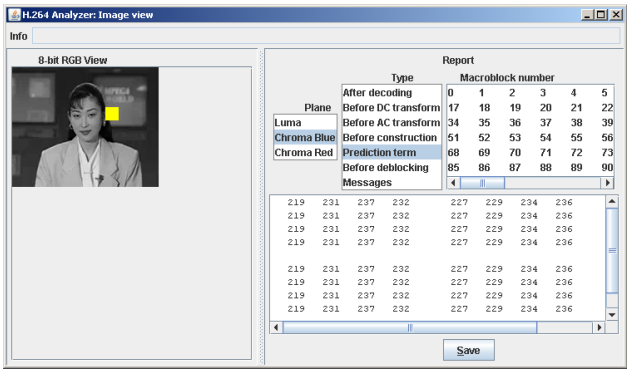

Fig.4 – GUI-based diagnostic tool: picture analysis.

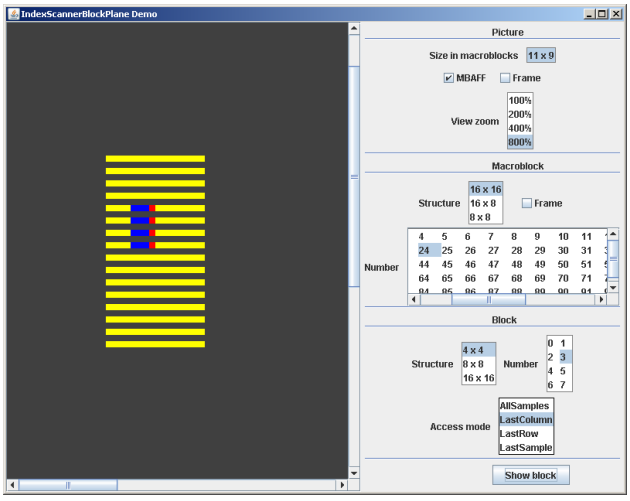

Fig.5 – A tool for demonstrating the PAFF and MBAFF modes of sample access.

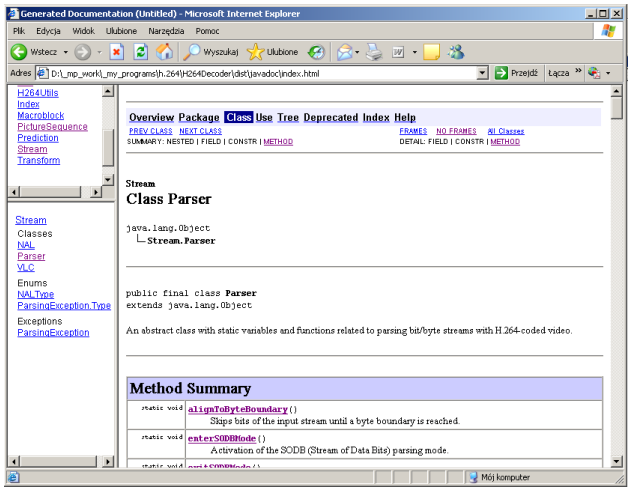

Fig.6 – Class documentation generated using JavaDoc.

7. CONCLUSION

Our case shows that having carefully selected tools and building a suitable development methodology upon them are essential for the success of a hi-tech project. After analyzing possible approaches to H.264 development, we promote Java as both programming language and entire technology that allows advanced DSP algorithms to be prototyped with high productivity. Using it we were able to quickly, in half a year, implement both a software H.264 decoder and accompanying tools, even though the team decided to do the work from scratch and did not specialize in video processing. The well-documented and well-organized code forms a basis for developing a high-performance real-time hardware implementation. The works are in progress, and results are expected soon.

8. ACKNOWLEDGEMENT

This work was supported by Bialystok Technical University under the grant W/WI/8/08.

9. REFERENCES

[1] ITU-T. ISO/IEC. *ITU-T Rec. H.264 Advanced video coding for generic audiovisual services / ISO/IEC 14496-10 MPEG-4 AVC*. ITU. Geneva 2003. [On-line]. Available: http://www.itu.int/rec/T-REC-H.264

[2] T. Wiegand. G. Sullivan. G. Bjontegaard. A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Trans. Circuits Syst. Video Technol.* 13 (7) (2003). p. 560-576

[3] D. Marpe. T. Wiegand. G.J. Sullivan. The H.264/MPEG4 Advanced Video Coding standard and its applications. *IEEE Commun. Mag.* 44 (8) (2006). p. 134-143

[4] S.-k. Kwon. A. Tamhankar. K.R. Rao. Overview of H.264/MPEG-4 part 10. *J. Vis. Commun. Image R.* 2 (17) (2006). p. 186-216

[5] R. Schäfer. T. Wiegand. H. Schwarz. The emerging H.264/AVC standard. *EBU Tech. Review* 12 (2003)

[6] I.E.G. Richardson. *H.264 and MPEG-4 video compression*. Wiley. Chichester, UK, 2003. p. 305

[7] C.S. Kannangara. *Complexity management of H.264/AVC video compression*. PhD Thesis. The Robert Gordon University, 2006

[8] Y. Chen. E. Li. X. Zhou. S. Ge. Implementation of H.264 encoder and decoder on personal computers. *J. Vis. Commun. Image R.* 17 (2006). p. 509-532

[9] H.264/AVC Reference Software. [Online]. Available: http://iphome.hhi.de/suehring/tml/

[10] ITU-T. ISO/IEC. *ITU-T Rec. H.264.2 Reference software for H.264 advanced video coding / ISO/IEC 14496-5 MPEG-4 Reference software*. ITU. Geneva 2001.

[11] Mailing list for x246 developers. [Online]. Available: http://mailman.videolan.org/listinfo/x264-devel

[12] Mp4-tech mailing list. [Online]. Available: http://lists.mpegif.org/mailman/listinfo/mp4-tech

[13] The FFmpeg libavcodec library. [Online]. Available: http://ffmpeg.org

[14] x264, a free H.264/AVC encoder [Online]. Available: http://www.videolan.org/developers/x264.html

[15] H264Visa. [Online]. Available: http://www.h264-visa.com

[16] M. Livshitz. M. Parfieniuk. A. Petrovsky. Wideband CELP coder with multiband excitation and multilevel vector quantization based on reconfigurable codebook, *Digital Signal Process. (OOO "KBWP", Moscow, Russia)* 2 (2005). p. 20-35

[17] A. Petrovsky. M. Parfieniuk. A. Borowicz. Warped DFT based perceptual noise reduction system *Proc. 116th AES Conv.* Berlin, Germany, 8-11 May 2004. Conv. Paper #6035

[18] J.A. Fisher. P. Faraboschi. C. Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann/Elsevier. San Francisco, CA, 2005. p. 712

[19] J. Labrosse. et al. *Embedded software: know it all*. Newnes/Elsevier. Oxford, 2007. p. 792

[20] C.D. Locke. P.C. Dibble. Java technology comes to real-time applications. *Proc. IEEE* 7 (91) (2003). p. 1105-1113

[21] P.C. Dibble. *Real-time Java platform programming*. Prentice-Hall. Englewood Cliffs, NJ, 2002. p. 352

[22] A. Wellings. *Concurrent and real-time programming in Java*. Wiley. 2004, p. 446

[23] J. Baker. A. Cunei. C. Flack. F. Pizlo. M. Prochazka. J. Vitek. A Real-time Java Virtual Machine for avionics: an experience report. *Proc. 12th IEEE Real-Time Embedded Technology Appl. Symp. (RTAS)*. San Jose, CA, 4-7 April 2006, p. 384-396

[24] A. Wellings. A. Burns. Real-time Java. *in Handbook of real-time and embedded systems*. Ed: I. Lee. J.Y. Leung. S.H. Son. Chapman & Hall/CRC. Boca Raton, FL, 2008. p. 12-1–12-19

[25] R. Gordon. S. Talley. *Essential JMF: Java Media Framework*. Prentice Hall. Upper Saddle River, NJ, 1999