VYA: Applying YAPI To Video

Version 1.4.0

Kees van Zon

Keywords: YAPI, application modelling, video, re-usability

Abstract: This document introduces VYA, a procedure for video-oriented use of YAPI, the Y-chart Application Programmer's Interface developed at Philips Research Eindhoven for modeling and simulating signal processing applications. VYA resulted from a cooperation between the developers of YAPI and video algorithm developers at Philips Research Briarcliff. VYA-compliant software modules are plugable by virtue of standardized interfaces for video and control signals, while a high level of re-usability is obtained through a carefully chosen application of YAPI. VYA focuses on communicating rectangular images in applications with YUV and/or RGB input and/or output.

Contents

1 Introduction

This document outlines a set of coding rules for video-oriented use of YAPI, the Y-chart Application Programmer's Interface developed at NatLab for modeling and simulating signal processing applications. These rules, referred to as "Video YAPI" or simply VYA, resulted from a cooperation between the developers of YAPI, whose field of expertise is system design tools, and video algorithm developers at Philips Research Briarcliff. As such, they meet the needs of video algorithm designers that wish to express their algorithms in a unified manner, as well as the needs of system developers wishing to efficiently implement such algorithms using a Y-chart based design flow. VYA compliant software modules are plugable and highly re-usable. The former is by virtue of standardized interfaces for video and control signals; the latter is obtained by a carefully chosen application of YAPI. As a result, a project that develops VYA modules can feed its output efficiently into projects that deal for example with:

- efficient hardware and/or software implementation of video algorithms
- the design of new architecture templates for video processing
- functional modeling and analysis of larger video processing systems.

2 YAPI

The purpose of YAPI is to provide a means for constructing high-level, functional software models (i.e. functional specifications) of signal processing applications in such a way that they [1]

- are reusable across projects,
- come with a run-time environment for execution on a workstation or PC,
- have a platform independent programmer's interface, and
- provide input for platform dependent mapping and performance analysis.

YAPI improves system design flows by separating application modeling from other system design activities such as architecture modeling, mapping, and performance analysis. It provides a means for structured design and modeling of signal processing applications by explicitly exposing parallelism and communication, and it

speeds up system design because the platform independent programmer's interface enables re-use [1].

YAPI is a C++ library with a set of rules for modeling signal processing applications as a process network consisting of processes that are interconnected by directed first-in-first-out channels called fifos. Communication is based on the Kahn Process Network model with blocking reads on theoretically unbounded fifos [2]. In practice, fifos are bounded so that writes can also suspend process execution. Processes have a private state space which is not accessible by other processes. They communicate with their environment through input and output ports which are the formal arguments of processes. Fifos connect output ports to input ports; they are the actual arguments of processes. Each port is connected to precisely one fifo. Processes can read from their input ports and write to their output ports. The read method blocks if the corresponding fifo is empty. The write method blocks if the corresponding fifo is full, which avoids data loss during communication. Ports and fifos are typed which means that they can handle data of one type only. The structure of process networks, i.e., the binding of fifos to ports, is static; it is known at compile time and does not change at run time [1].

3 Specification Level

VYA defines interfaces for YAPI processes or (possibly nested) process networks with Y,U,V and/or R,G,B inputs and/or outputs. Any process or process network that adheres to these interfaces and to the preferred coding style is called a VYA module. The granularity of a module can range from small (e.g. gamma correction, look-up table) through medium (e.g. scan rate converter) to large (e.g. video processing chain). VYA modules can be combined into VYA systems, and they can be embedded in non-VYA YAPI systems.

4 Video Signal Types

This version of VYA defines interfaces for Y,U,V and R,G,B processing only. As such, the functional categories that it covers are source decoding (output only), image enhancement and display adaptation. In practice, some modules will process a single video signal only, e.g. Y, and others will process multiple video signals, e.g. U,V or R,G,B. Furthermore, some ports will be dedicated to a specific signal type, e.g. Y, whereas others ports may allow multiple signal types, e.g. Y,U,V,R,G,B. VYA therefore allows modules to have any number and type of video signals at their input ports and at their output ports. Some typical examples are shown in

Table 1.

input	output	example
(CVBS)	Y,U,V	NTSC decoder
(TS)	Y,U,V	MPEG2 decoder
Y	Y	LTI
U,V	U,V	CTI
Y,U,V	Y,U,V	scan rate conversion
Y,U,V	R,G,B	matrix
R,G,B	R,G,B	contrast
C	C	generic spatial scaler (C = Y or U or V or R or G or B)

Table 1: module input/output examples

5 Video Communication Data Type

5.1 Introduction

Video signals are communicated between processes by means of tokens that travel over fifo channels. Modules are seamlessly plugable when there is a match between the number of I/O ports, the signal types (e.g. Y,U,V) communicated through these ports, and the data types of the tokens that make up each signal. The choice of the latter, which is referred to as the communication data type, determines the execution speed, the memory usage, and the re-usability of YAPI code [1],[3]. Our primary goal is maximum re-usability, which requires that the application defines exactly what is communicated, but not how: it is the mapping of a specification to an implementation that decides how the communication is done most efficiently. This goal is supported by communicating the smallest possible token size, which, for video processing, is a pixel. Communicating pixels allows the application programmer to choose his processing to be on a pixel, line, field or frame basis, simply by reading and writing vectors of pixels. YAPI supports efficient vector read/write methods for this purpose [1].

5.2 Definitions

All VYA definitions reside in a file called vya.h; they are step by step introduced below. To begin with, video pixels are of type VYAsigned16bit:

```
typedef signed16bit VYApixel;
```

The implementation of type VYAsigned16bit is a design decision which is independent of VYA and must be specified in a separate file called architecture.h. For example:

```
typedef short VYAsigned16bit;
```

In computations on type VYApixel, VYA assumes signed data ranging from -215 to 215-1. In all implementations, type VYAsigned16bit must therefore contain at least 16 bits. See also section 8. Vectors of pixels either represent one-dimensional arrays called lines or two-dimensional, rectangular arrays called images. The number of pixels in a line is indicated by VYAlineLength. As it may vary for each line that is communicated, VYAlineLength must be supplied for each line.

```
typedef unsigned int VYAlineLength;
```

The size of an image is defined by VYAimageWidth which indicates the (fixed) number of pixels per line, and VYAimageHeight which is the number of lines per image. Both must be supplied once for each image. Finally, an image can be either a top field, a bottom field, or a frame. This is indicated by VYAimageType, to be supplied per image when needed by the application.

```
typedef unsigned int VYAimageWidth;
typedef unsigned int VYAimageHeight;
enum VYAimageType {VYA_TOP_FIELD,VYA_BOTTOM_FIELD,VYA_FRAME};
```

5.3 Interfaces

The re-usability of any YAPI process or process network improves when it communicates only information that is required for proper operation. A gamma correction process for instance operates on individual pixels and does not need the notion of lines and images; it should therefore not receive any VYAlineLength, VYAimageWidth, VYAimageHeight or VYAimageType tokens. Similarly, a horizontal filter does not need the notion of images, etc. VYA therefore distinguishes between pixel-based, line-based and image-based input and/or output interfaces, as shown in Figure 1 - Figure 3.

Notes:

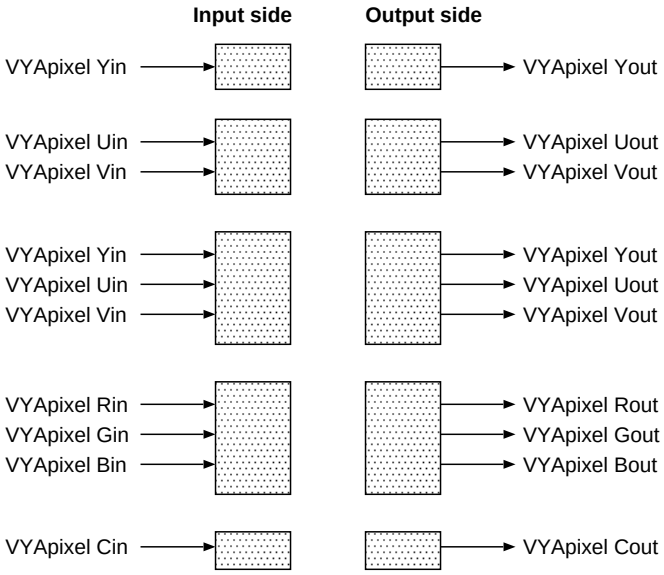

Figure 1: VYA interfaces for modules with pixel-based input and/or output

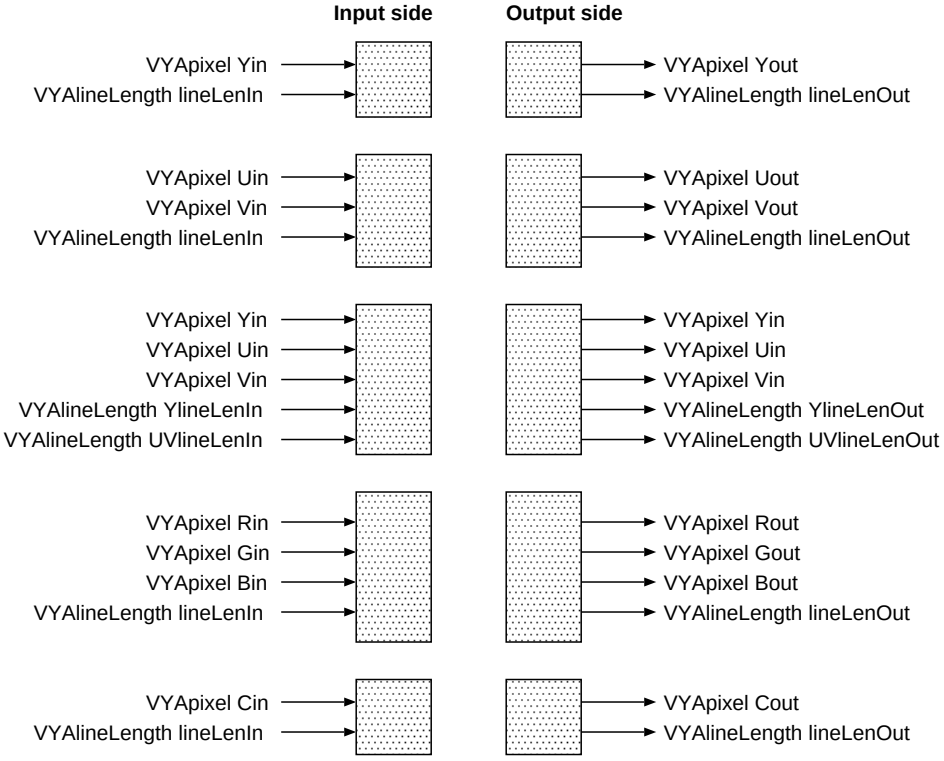

Figure 2: VYA interfaces for modules with line-based input and/or output

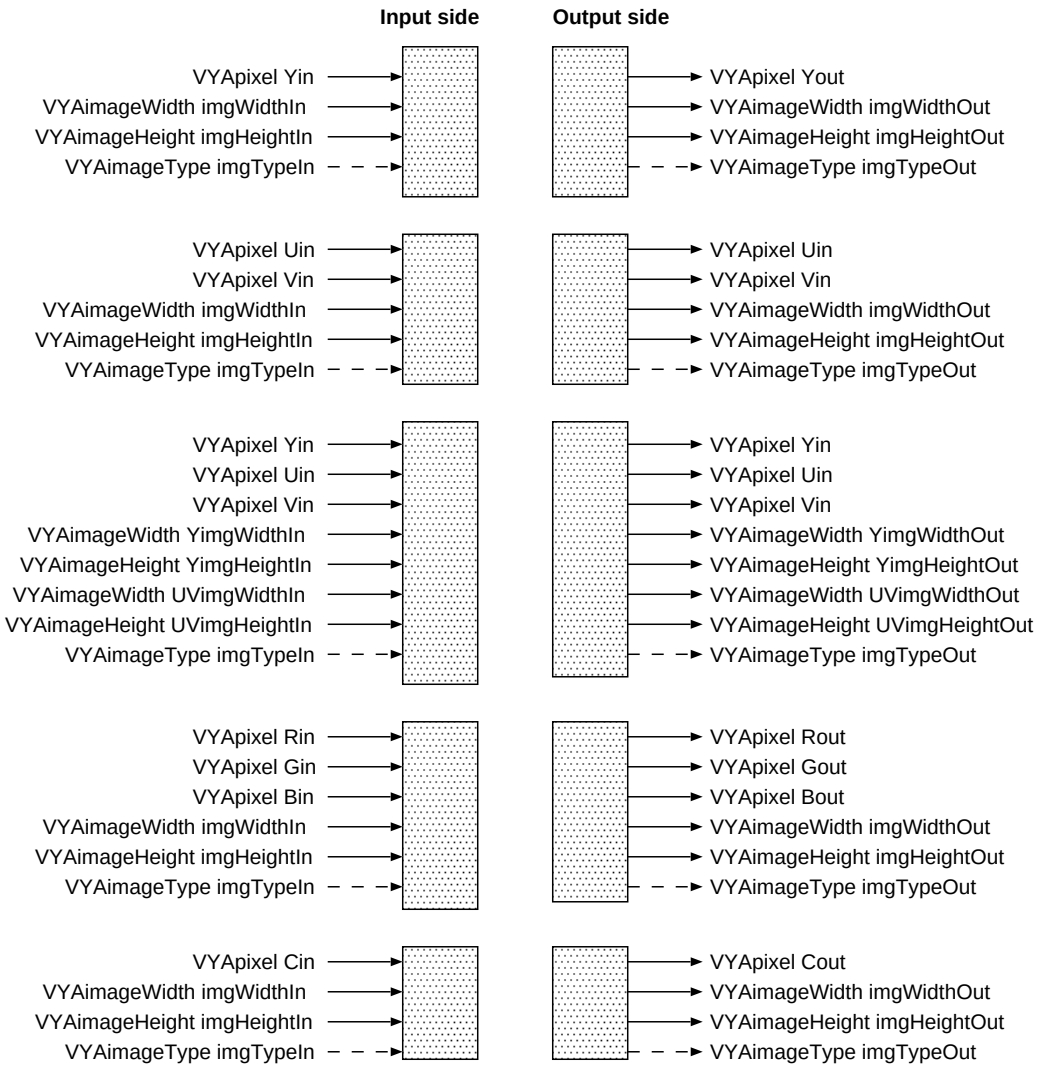

Figure 3: VYA interfaces for modules with image-based input and/or output

- ”C” stands for ”Component” and indicates that the interface can be used for any signal type.
- For line-based modules with both U and V inputs or outputs, it is assumed that these signals have the same line length.
- For image-based modules with both U and V inputs or outputs, it is assumed that these signals have the same image size and image type.
- For image-based modules with Y, U and V inputs or outputs, it is assumed that all three signals are of the same image type; Y and U,V may be of different size, however.
- For image-based modules with R, G and B inputs or outputs, it is assumed that all three signals are of the same size and image type.
- Output interfaces for line length, image size and image type should only be provided for those parameters that can be modified by the module.
- Modules may have additional interfaces, e.g., for control purposes.

5.4 Conversions

A system composed of VYA modules needs converters when it contains both modules with line-based and modules with image-based interfaces. These converters are shown in Figure‘4 below. Module ”i2l” converts VYAimageWidth to VYAlineLength by creating VYAimageHeight output tokens for each input token it receives. Likewise, module ”l2i” converts VYAlineLength to VYAimageWidth by reading VYAimageHeight input tokens and producing a single output token.

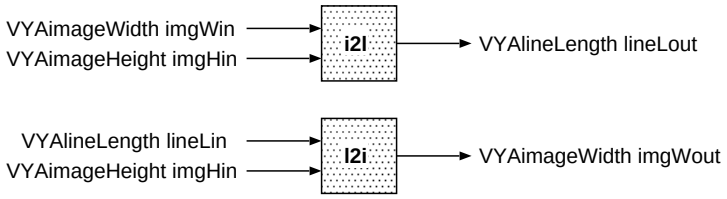

Figure 4: VYA modules for converting between line-based and image-based interfaces.

Figure 5 shows an example application of ”i2l”. Note that it reveals a need for multicasting, i.e., connecting multiple fifos to a single output port. This will be supported by later YAPI versions.

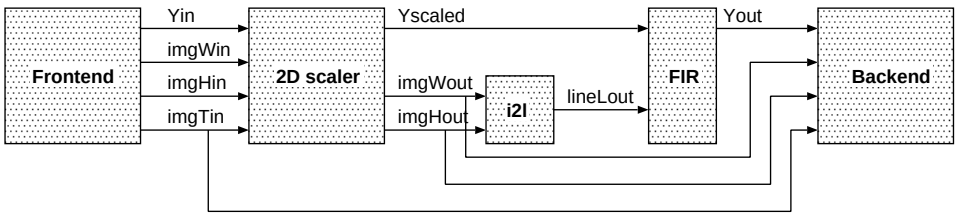

Figure 5: Example application of image-to-line conversion.

6 Video Processing Data Type

The video processing data type refers to the data entity on which a module operates internally. Typically, this is a pixel, a line, a field, or a frame, but it can for instance also be a vertical stripe or a two-dimensional block. VYA poses no restrictions on the video processing data type; it only requires that the external interfaces of a module adhere to the rules outlined in the previous section.

7 Image Resolution

Given applications like dual window and PiP, the introduction of the ATSC standard with its multitude of new image sizes, and the emergence of flat panel displays with various fixed resolutions, VYA modules should preferably be capable of handling any image size. When modules receiving lines or images have restrictions in the sizes they can handle, they must call function VYAcheckSize(), which checks if the module can actually process the line or image size it receives and terminates execution with a printed error message if it can't. VYAcheckSize() also manages the input buffers and is defined in vya.h,cc as a member function of class VYAprocess.

8 Bit Precision

The optimum choice for the bit precision of the video signals of a video processing algorithm may differ for each particular implementation of that algorithm. The re-usability of an application model therefore improves when it allows a range of bit precisions. In support of this, VYA requires that the bit precision of the video input and the video output signals of each model is parametrized. An adverse impact on the simulation speed is prevented by adopting a very simple mechanism for this purpose: type VYApixel is assumed to range from -2^{15} to 2^{15-1}, while the

effective range of the video input and/or output pixels is defined by parameters. To reflect that the bit precision of an implementation is fixed, the bit precision parameters are passed as arguments in the constructor of a process (network), which allows them to be programmable yet forces them to be constant during each instantiation. For example:

```
class myVYAprocess : public VYAprocess
{
public:
  // constructor
  myVYAprocess(
    Id                n,
    VYAbitPrecision   inPrec,
    VYAbitPrecision   outPrec,
    ...
};
```

Class VYAbitPrecision is defined in vya.h,cc as follows:

```
class VYAbitPrecision
{
public:
  // constructors
  VYAbitPrecision();
  VYAbitPrecision(VYAbitPrecision&);
  VYAbitPrecision(unsigned int, bool);
  // member variables
  unsigned int  bits;
  bool          signbit;
};
```

Table 2 explains how the bit precision parameters determine the pixel range. These parameters must be supplied for input as well as output video interfaces. When a module has multiple video input and/or output interfaces, the programmer can choose to provide separate parameters for each interface or to combine them into a single parameter - e.g., UbitPrecIn and VbitPrecIn vs. UVbitPrecIn. To prevent incorrect behavior caused by illegal parameter values (e.g. signbit = true for a Y interface), modules must check parameter correctness by calling function VYAcheckPrecision which is a member function of VYAprocess provided in vya.h,cc.

The internal precision of an algorithm must be sufficient to handle the maximum input pixel range indicated in Table 2. Generally, it will suffice to use a 32-bit data type like int for intermediate results. Final results must be normalized to the required output precision; for this purpose, vya.h provides inline functions for rounding, shifting, clipping etc.

signal type	U,V	Y,R,G,B
signbit	true	false
pixel range	$-2^{\text{bits}-1}...2^{\text{bits}-1}-1$	$0...2^{\text{bits}-1}$
bits range	VYA_MIN_BITS...VYA_MAX_BITS_U	VYA_MIN_BITS...VYA_MAX_BITS_S
VYA_MIN_BITS range	1...VYA_MAX_BITS_U	1...VYA_MAX_BITS_S
VYA_MAX_BITS_U range	VYA_MIN_BITS...16	
VYA_MAX_BITS_S range		VYA_MIN_BITS...15

Table 2: Definition of effective bit precision. All constants reside in vya.h

Because mapping an algorithm to an efficient implementation is facilitated by understanding its internals, the internal precision of a VYA module is preferably indicated by means of comments. In the example shown below, a filter with gain 29 is applied to a luminance line. By definition, the overall gain of a module equals $2^{\text{outPrec.bits-inPrec.bits}}$, which implies that the filter output must be normalized by dividing it by $2^{9-\text{outPrec.bits+inPrec.bits}}$ using round and shift operations. Luminance data being unsigned, the result must finally be clipped between 0 and 2outPrec.bits-1 to ensure the correct output range. Comments indicate the bit precision of the input data and of the result of every processing step. The format used is s.n.m, where, s indicates the sign bit (0: positive; 1: negative; s: positive or negative), n indicates unused bits, and m indicates magnitude bits. Besides 16 bits for type VYApixel, this example assumes 32 bits for type int.

```
void myVYAprocess::processLine()
{
  int min, max, sf, data;
  unsigned int pix;

  min = 0;
  max = (1 << outPrec.bits) - 1;
  sf = inPrec.bits - outPrec.bits; // scale factor

  for (pix=0; pix<imgWidth; pix++) {
    // read pixel
    data = (int) Yin[pix];         // {0.31-inBits.inBits}

    // insert processing here

    // round, normalize and clip
    data = RNORM(data, sf);        // {0.31-outBits.outBits}
    data = CLIP(data, min, max);   // {0.31-outBits.outBits}

    // write pixel
    Yout[pix] = (VYApixel) data;   // {0.15-outBits.outBits}
  }
}
```

9 Control

Video processing algorithms frequently offer - or require - some form of external control, which is for our purpose simply defined as being any kind of non-video data. VYA distinguishes between four types of control parameters:

- constants
  - defined in the code at compile time
  - cannot change unless the code is modified and re-compiled
  - examples: VYA_MIN_BITS, VYA_MAX_BITS_S, VYA_MAX_BITS_U
- parameters
  - defined as arguments in the constructor of a process (network) at instantiation time
  - cannot change during an instantiation of a process (network)
  - example: bitPrec
- deterministic tokens
  - defined through input ports of a process (network) at run time
  - used to control the data flow between modules, e.g. the amount of data consumed and/or produced, or the routing of data
  - may change during an instantiation of a process (network), but must be synchronized with tokens that control modules with which the module in question interacts; improper synchronization would lead to incorrect system behavior
  - example: lineLen, imgWidth, imgHeight
- non-deterministic tokens
  - defined through input ports of a process (network) at run time
  - used to control the data contents rather than the data flow
  - may change at any moment during an instantiation of a process (network) without affecting proper behavior of the module
  - reading is conditioned by YAPI's select function [1]
  - example: filterCoefficients

10 Deadlock

VYA does not specify the length of the fifos that connect modules. In [3] it was shown that bounded fifos with default sizes may lead to deadlock situations caused by a blocking write action. As of v0.5, YAPI's run-time environment automatically increases fifo sizes at run-time when this situation is detected. As a result, deadlock now means that all processes are blocked by a read action. The upper limit on the total size of the growing fifos depends on the amount of memory available in the system.

11 VYA Systems

Creating a video processing system from VYA modules requires frontend and backend modules which produce and consume video data, respectively. The video data itself may be stored in files of various formats or come from a real-time source; each option requires a specific frontend/backend. Video processing modules should always be independent of any frontend and backend, i.e., no information specific to a particular frontend/backend should be communicated to these modules. The frontend and backend are controlled by a subsystem consisting of one or several control processes. This control subsystem may for instance read the system configuration from initialization files and send optional, module-specific control parameters to the video processing modules at the proper data rate. An example is shown in Figure 6, in which dataX symbolizes video channels, sizeX indicates channels for communicating VYAlineLength, etc., and inSeqInfo and outSeqInfo are channels for communicating information about the input and output video sequences. The control subsystem receives an imgSync token for synchronizing control parameters with the video. As indicated by the arrows, the flow of control parameters may be bidirectional.

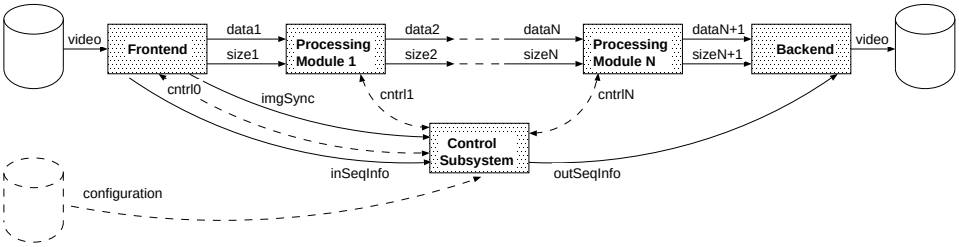

Figure 6: VYA system with video chain and control subsystem. Dashed lines are optional.

12 Coding Style

A uniform look and feel improves the re-usability of code, and a well-defined coding style helps achieve this. VYA therefore adopts the coding style set forth in [1] with some additions as outlined below. Regarding the naming conventions: names of files, variables, types, etc., start with lower case letters. Multiple words within a name are separated by upper case letters; e.g. nrPixelsPerImage. Exceptions may make sense, e.g. YinData. All processes with video i/o must be derived from class VYAprocess which is provided in vya.h,cc.

12.1 File Names

Following [1], every process and every process network must reside in a separate file. The same name is used for the process (network) and the corresponding .cc and .h files, although the file names use lower case only:

- files myvyaprocess.cc,h define process myVYAprocess
- files myvyaprocessnetwork.cc,h define process network myVYAprocessNetwork

12.2 Variable Names

The choice of variable names greatly impacts the readability of code. The meaning of c = a * b; is much less obvious than the meaning of nrPixelsPerImage = imgWidth * imgHeight; VYA therefore requires the use of clear and descriptive names for variables, ports, etc. When long names are abbreviated, the meaning of the abbreviation must be explained with a comment.

12.3 Constant Names

Names of constants are in capitals, e.g. PI. Multiple words within a name are separated by underscores, e.g. TWO_PI.

port type	base name
Video in	Yin, Uin, Vin, Rin, Gin, Bin, Cin
Video out	Yout, Uout, Vout, Rout, Gout, Bout, Cout
Line length in	lineLenIn, YlineLenIn, UVlineLenIn
Line length out	lineLenOut, YlineLenOut, UVlineLenOut
Image width in	imgWidthIn, YimgWidthIn, UVimgWidthIn
Image width out	imgWidthOut, YimgWidthOut, UVimgWidthOut
Image height in	imgHeightIn, YimgHeightIn, UVimgHeightIn
Image height out	imgHeightOut, YimgHeightOut, UVimgHeightOut
Image type in	imgTypeIn
Image type out	imgTypeOut

Table 3: Base names of standard I/O ports

parameter	name
port (formal)	baseNameP
fifo (actual)	baseNameF
variable	baseName

Table 4: Parameter names

12.4 Port Names

Port names are standardized as shown in Figure 1 - Figure 3. From these so-called ”base names” which are repeated in the table below, the names of the actual parameters, formal parameters and variables of a process or process network are derived. Table 4 shows this for a port called baseName. Variables may be given more appropriate names where applicable.

12.5 Fifo Names

The fifos of a process network should indicate the type of data traveling through it. VYA’s base names for standard fifos are shown below. When there is more than one fifo of the same type, the base names are followed by a number.

12.6 Process and Process Network Names

Processes and process networks must be given descriptive names.

fifo type	base name
Video	Y, U, V, R, G, B, C
Line length	lineL, YlineL, UVlineL
Image width	imgW, YimgW, UVimgW
Image height	imgH, YimgH, YimgW
Image type	imgT

Table 5: Base names of standard fifos

12.7 Comments

Statements or groups of statements that are not completely obvious should be explained by means of comments. As outlined above, this includes the bit precision of mathematical operations.

12.8 Headers

All files representing VYA processes and process networks should be preceded by the header shown below. Fields that don't apply to a particular file are designated "n/a". The headers may be automatically generated by the version control system so that they are always up to date.

12.9 Layout

The layout of a program significantly impacts its readability. Brackets delimiting flow statements (for, while, if, ...) should be properly aligned. Comments should follow or be vertically aligned with the statement(s) they refer to. Spaces should be used instead of tabs to prevent the layout from being messed up in editors with different tab settings.

12.10 Error Signalling and Diagnostics

The number of print statements in VYA code should be kept to a minimum. Print statements inserted for debugging purposes during module development should be removed prior to acceptance into a library. Two techniques are used for signalling errors caused by improper application of a VYA module in a larger system. Errors in low-rate control parameters are signalled with unconditional print statements,

e.g. as in the VYAcheckSize and VYAcheckPrecicion functions. Errors in pixel-rate video data are optionally checked with the assert function, for instance

```
assert(!((Yin[i] < 0) || (Yin[i] > Ymax)));
```

To preserve simulation speed, assert should by default be turned off by defining compiler directive -DNDEBUG in the makefile. For diagnostic purposes, the makefile can also enable a verbose mode through optional compiler directive -DVERBOSE, to be used for instance as follows:

```
#ifdef VERBOSE
  cout << fullName()
       << ": input line length = "
       << lineLengthIn
       << endl;
#endif
```

The fullName function should always be included to identify the issuing process.

12.11 Miscellaneous

Additional coding rules for writing re-usable DSP algorithms are given in [4]. They include:

- Static member variables may not be used, as they are not duplicated for multiple instances and hence cause dangerous side effects.
- Pointers may not be communicated over fifos, as that assumes a shared memory architecture.
- No hard paths to include files may be used, as that limits portability. All paths should be relative to the directory where the module is located.
- Type casts are explicit operations and should preferably be explicit.
- Floating point variables should be used sparingly, as both hardware and software implementations are most likely to use fixed point.
- File I/O is only allowed by frontend, backend and control modules, except for print statements in the verbose mode which any module may have.

13 Test Bench

To assist users of a VYA module in understanding how it is to be applied, a simple example is to be provided in the form of a process network containing a frontend module, a backend module, a control module, and the VYA module in question. The frontend and backend should be links to standard VYA modules, e.g., for I/O of PFSPD files. This test bench should be accompanied by the makefile, initialization files where applicable, video test sequences, and the expected response to these sequences to allow module behavior to be verified. Preferably, a typical and an extreme sequence are provided for illustrating both normal and exceptional behavior. Test sequences should be as short as possible to minimize storage space and transmission bandwidth.

14 Documentation

All VYA modules should be accompanied by a document that contains at least the following, as this is important for their re-usability:

1. Introduction

- brief outline of function; mention of existing implementations, e.g., IC type number

2. Feature list

- overview list of all features (data book style)

3. Algorithm

- brief description of algorithm; use references for more detail

4. System overview

- partitioning of module into processes
- fifo names (input, output, internal)
- indication of memories

5. Performance

- image quality
- compute resources (execution speed, memory usage)

6. Interface

- video I/O: data type, port names, token size, data rate, image sizes
- control: data rate, type (user control vs. system control)

7. Future work

- suggestions and recommendations, e.g., for algorithmic improvements

8. References

Appendix A. code structure

- list of directories and files

Appendix B. example

- brief description of the example system supplied with the module, including block diagram and test sequence(s)

15 Directory Structure

In support of re-use and maintainability, a VYA module's code files and accompanying documents are to be stored in a standard directory tree named after the module (e.g. mymodule), as indicated in Table 6.

The (partial) directory tree of a VYA library with version control could for instance look like follows (with "$(HOME)" the directory in which the modules are installed):

$(HOME)/Modules/vya-0.2/
$(HOME)/Modules/vya-0.3/
$(HOME)/Modules/vya-0.3/cvbs_decoding/
$(HOME)/Modules/vya-0.3/front_backends/
$(HOME)/Modules/vya-0.3/front_backends/sgi_divo/
$(HOME)/Modules/vya-0.3/front_backends/sgi_divo/v1/(as in Table 6)
$(HOME)/Modules/vya-0.3/front_backends/vya_pfspd/
$(HOME)/Modules/vya-0.3/front_backends/vya_pfspd/v1/(as in Table 6)
$(HOME)/Modules/vya-0.3/mpeg_decoding/
$(HOME)/Modules/vya-0.3/noise_reduction/
$(HOME)/Modules/vya-0.3/picture_control/
$(HOME)/Modules/vya-0.3/scanrate_conversion/
$(HOME)/Modules/vya-0.3/sharpness_enhancement/
$(HOME)/Modules/vya-0.3/spatial_scaling/

mymodule/	subdirectory contents	remarks
doc/	module documentation, referenced docs (optional)	
example/	myvyamoduleexample.cc, myvyamoduleexample.h, myvyamoduleexample.main.cc, makefile, video test sequences and responses (opt.), initialization files (opt.)	- video sequences may be referenced through links - frontend and backends referenced in makefile - makefile makes myvyamoduleexample
include/	myvyaprocess1.h, myvyaprocess2.h, ... myvyamodule.h	myvyamodule is the module's process network (opt.)
lib/ Makefile	libmyvyamodule.a	subdirectory per O.S. makes lib/ and obj/
obj/	myvyaprocess1.o, myvyaprocess2.o, ... myvyamodule.o	subdirectory per O.S.
readme.txt		file-header-like module info plus installation instructions
src/	myprocess1.cc, myprocess2.cc, ... myvyamodule.cc	hierarchical for nested process networks

Table 6: VYA directory tree (in alphabetical order)

16 Conclusion

We have presented VYA, a procedure for applying YAPI to video which can significantly enhance the re-usability of video processing algorithms. VYA is limited to applications with YUV and/or RGB input and/or output. Other signal types can be added in future versions. Appendix A provides a compliance check list; Appendix B provides a detailed example.

Acknowledgments

The author is grateful to Erwin de Kock, Gerben Essink, Karl Wittig, Nehal Dantwala, Ralph Braspenning, Walid Ali, Wim Smits and Yibin Yang for their contributions to the VYA standard, and to Rob Vogelaar and Dave Bryan for reviewing this document.

References

[1] E. de Kock, G. Essink, *Y-chart Application Programmer's Interface*, Philips Research Eindhoven Technical Note 008/99, March 1999.

[2] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, Proceedings of IFIP Congress 74, North-Holland Publishing Co., 1974.

[3] K. van Zon, *VYA: A Proposal For Applying YAPI To Video - A Discussion Document*, Philips Research Briarcliff DTV/UTV Report 004, v.02, August 1999.

[4] K. van Zon, *Writing Re-usable DSP Algorithms*, Philips Research Briarcliff Technical Note 2000-07, January 2000.

Appendix A Compliance Check List

- video input and/or output signals are a subset of Y,U,V and R,G,B
- video communication data type is VYApixel
- pixel-based modules receive/produce no size information
- line-based modules receive/produce line length
- image-based modules receive/produce image size and image type (opt.)
- line length, image size and image type outputs only provided when modified by module
- input line length or image size is parameterized and checked by checkSize
- input and output bit precision is parameterized and checked by checkPrecision
- internal bit precision can handle 16-bit input range and is indicated by comments
- video output signals are properly rounded, normalized and clipped
- no frontend/backend specific information is passed to/from the module
- names of files, processes, variables, ports and fifo's are clear and descriptive
- names of ports and fifos follow Table3 - Table 5 where applicable
- code is clarified by comments as needed
- all files contain up-to-date headers
- the code layout is neat and contains no tabs
- print statements are only for signalling errors or for diagnostic purposes
- signalling of errors in low-rate control parameters uses unconditional print statements
- signalling of errors in pixel-rate video data is through the assert function
- signalling for diagnostic purposes is conditional to compiler directive VERBOSE
- assert is default turned off by defining compiler directive -DNDEBUG in the makefile
- no static variables are used

- no pointers are communicated over fifos
- code files contain no hard paths to other files
- type casts are explicit
- floating point variables are used only when needed
- modules other than frontend, backend and control perform no file I/O
- a test bench with frontend, backend and control modules is provided along with a makefile
- video test sequences and their responses are provided
- a document with the framework outlined in Section 14 is provided
- all files are stored in a directory tree that follows Section 15