IMEM: An object-oriented memory- and interface modelling approach for real-time video processing systems

Benny Thörnberg, Håkan Norell and Mattias O'Nils
Mid Sweden University, Dept. of Information Technology and Media, Sweden
Phone: +46 60 148600, E-mail:{bentho|hakan.norell|mattias}@itm.mh.se

Abstract

Most operations invoked in video processing systems are neighbourhood oriented. For a video system designer, this limited spatio-temporal collection of pixels represents a natural abstraction. In this paper, we present a basic set of object-oriented design entities. Entities, which can be combined to capture an interface- and memory model at a conceptual level, with the neighbourhood as an abstraction. These design entities, called IMEM, are implemented as an extension to SystemC. IMEM supports conceptual modelling that excludes implementation details and has explicit data dependency built-in to the model. This makes IMEM a very efficient starting point for design-space exploration and system synthesis. A spatio-temporal noise-reduction filter is selected as a test-vehicle within a case study. This filter is captured using both IMEM and a standard SystemC modelling-style. The simulation performance and the modelling efficiency are compared. This comparison shows that IMEM is about 50% more modelling efficient than a standard SystemC modelling-style. This increased efficiency comes to the cost of a 23% increase in simulation time.

Keywords: Interface- and memory modelling, SystemC, video systems, neighbourhood,C++

1 INTRODUCTION

Typical image processing operations [5] such as convolution, histogram, spatial and grey-level transforms, erosion, dilation and component labelling are all 2-D neighbourhood oriented. Consequently spatio-temporal Video Processing Systems (VPS) will operate on a 3-D neighbourhood [1][10], thus increasing the system complexity. From a VPS designer's point of view, the today's specification methods lacks in abstraction. The stream oriented abstraction chosen for the PHIDEO system [4] does not reveal the neighbourhood that naturally is common for most VPS operations. Nested loops, such as in a DFL-specification [8], need processing in order to analyse the data dependency between neighbourhood pixels. However, this analysis does not clearly separate the spatial and temporal mapping from the functional mapping of a video processing algorithm. VPS specifications written as nested loops define how the neighbourhood slides within a spatial domain. This is implementation related information which is an undesirable input during early design exploration.

Real-time video processing systems are data dominated. Typically the design bottleneck will be the memory data transfers maintaining a spatio-temporal neighbourhood. Another closely related and also critical design parameter is the large amount of background memory and the power dissipation coming from the high-speed accesses. These critical parameters have been addressed in [11] and an implementation using a memory hierarchy to overcome the memory access bottlenecks has been presented by Oelmann et al. [12]. Wuytack et al. [6] presents a more general methodology, where data reuse exploration is done by introducing application specific cache memory hierarchies. Applied on realistic VPS applications, the system design exploration tool ATOMIUM has enabled power reduction of about 90% [7]. ATOMIUM is based on both loop transformations and memory organisation decisions. Typically this exploration is done early in the design process. The ATOMIUM design entry is a DFL-specification [8], which needs additional profiling in order to extract inter-pixel and inter-frame data dependencies. The evolution of object-oriented specification methods based on class libraries has made language extensions possible to implement without having to update the compilers or simulators. Our previous research [13] indicates that SystemC [2] is a good candidate for modelling VPS.

Although some research has been made in the area of memory modelling and VPS, up until now, no research has shown the potential of combining a video designer friendly neighbourhood abstraction, conceptual modelling and early design space exploration methods into one homogeneous C++ system design environment. An environment that will take a VPS all the way from specification down to implementation. The reduction of time to market and the implementation optimisation serve as motivation for this and future research in this area.

This paper presents an object-oriented approach to conceptual memory- and interface modelling, called IMEM, that targets real-time VPS. Basic modelling entities such as input and output video stream-ports, frames, frame buffers and sliding bodies, provide the VPS designer with a specification method, that can easily capture stream ports, spatial and temporal mappings of video processing algorithms. This conceptual-level modelling excludes implementation details and provides the designer with means for explicit data-dependency modelling. Consequently, no additional analysis is needed to extract the data dependency, as in the case of nested loops. This will of course simplify the implementation of design space exploration methods. The entities are implemented in a SystemC class-library-extension and can be simulated together with standard SystemC modules [2]. The implementation of methods for early design exploration using loop transformations, data reuse analysis and memory hierarchy mappings is left for future research and is no longer considered in this paper.

The same test vehicle, a spatio-temporal median video-filter, as used in the SystemC-Ocapi comparison [13], is used to compare the simulation performance and modelling efficiency for both a standard SystemC modelling-style and using IMEM.

The rest of this paper is organised as follows. Section 2 explains how several IMEM modules can be combined into a signal flow graph in order to capture a complex real-time video processing system. Section 3 defines the basic design entities that can be combined to capture a memory and interface model. Section 4 explains how the basic entities can be combined using the test vehicle as an example. Section 5 summarises the results derived from a comparison of simulation performance and modelling efficiency. Finally section 6 concludes this paper.

2 IMEM MODELLING

Figure 1 shows three IMEM modules, two input- and two output video streamers linked together in a signal flow graph. The implementation of the connectivity for the simulation model for all these modules conform with the semantics of the Remote Procedure Call that comes with the SystemC Master-Slave Communication library [3]. The principles for denoting concurrent and slave processes in Figure 1, also conform with [3]. The functionality of each single IMEM module is defined according to the rules defined in section 3.

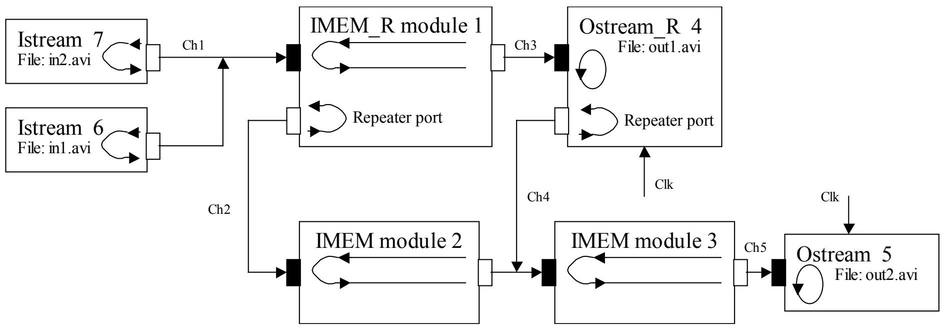


Figure 1. Simulator implementation signal flow graph with IMEM modules.

There exist two versions of an IMEM module, IMEM and IMEM_R. The latter has a repeater port that is used when one output video stream is feeding data to several input video master ports, thus it is used to model parallel connectivity. This mechanism allows IMEM module 2 in Figure 1 to access both input streaming modules the same way module 1 can. IMEM modules 2 and 3 are an example how two modules can be connected in series. The output streamer module has also an optional repeater port. The inter-module communication is based on an abstract protocol that provides the necessary arbitration of multi-port connections. This way, a single channel and a single port, can carry several inter-connecting video streams. The signal flow graph depicted in Figure 1 corresponds to the source code shown in Figure 2. The source code is described using the line numbers that are shown to the left in Figure 2. Line 3 instantiate the channels used for inter-module communication. The channels transfer *vtoken<int>*, which is a structured data type, carrying video data of type *int* and the abstract interconnecting protocol. Line 7 to 21 instantiate all modules. Every module is assigned its own unique module number. Line 25 to 36 connects module input ports, output ports and repeater ports together through channels. Line 42 to 46 initiates interconnecting video streams and 38 to 41, input/output streams assigned to disc files for simulation. This connectivity mechanism is important in order to support IP-component encapsulation, that is, the functionality of a single video processing operation can be defined without knowing its external context.

```
1   // Links
2
3   sc_link_mp<vtoken<int> > ch1,ch2,ch3,ch4,ch5;          // Channels with abstract protocol vtoken<int>
4
5   // Component instantiation
6
7   ovstream_r vout4("Video_out1");
8   vout4 == 4;
9   ovstream vout5("Video_out2");
10  vout5 == 5;
11  ivstream vin6("Video_in1");
12  vin6 == 6;
13  ivstream vin7("Video_in2");
14  vin7 == 7;
15
16  op1<int> operation1("Operation1");     // Video processing operation 1
17  op1 == 1;
18  op2<int> operation2("Operation2");     // Video processing operation 2
19  op2 == 2;
20  op3<int> operation3("Operation3");     // Video processing operation 3
21  op3 == 3;
22
23  // Connectivity description
24
25  vin6.ovport(ch1);                      // Channel 1 connects vin7 out, vin6 out and op1 input
26  vin7.ovport(ch1);
27  op1.ivport(ch1);
28  op2.ivport(ch2);                       // Channel 2 connects op1 repeater to op2 input
29  op1.rport(ch2);
30  op2.ovport(ch4);                       // Channel 4 connects op2 out, op3 in and vout repeater
31  op3.ivport(ch4);
32  vout4.rport(ch4);
33  vout4.ivport(ch3);                     // Channel 3 connects op1 output to vout in
34  op1.vout(ch3);
35  vout5.ivport(ch5);                     // Channel 4 connects op3 output to vout5 in
36  op3.ovport(ch5);
37
38  vout4.initStream(1,1,&ovideo1,NON_INTERLACED);        // Op1, stream 1, is source for output stream ovideo1
39  vout5.initStream(3,1,&ovideo2,NON_INTERLACED);        // Op3, stream 1, is source for output stream ovideo2
40  vin6.initStream(1,&ivideo1,NON_INTERLACED);           // Istreamer 6, has ivideo1 as source for stream 1
41  vin7.initStream(1,&ivideo2,NON_INTERLACED);           // Istreamer 7, has ivideo2 as source for stream 1
42  op1.initStream(1,6,1);                                // Istreamer 6, stream 1, is source for op1 input stream 1
43  op1.initStream(2,7,1);                                // Istreamer 7, stream 1, is source for op1 input stream 2
44  op2.initStream(1,7,1);                                // Istreamer 7, stream 1, is source for op2 input stream 1
45  op3.initStream(1,2,1);                                // Op2, stream 1, is source for op3 input stream 1
46  op3.initStream(2,1,1);                                // OP1, stream 1, is source for op3 input stream 2
```

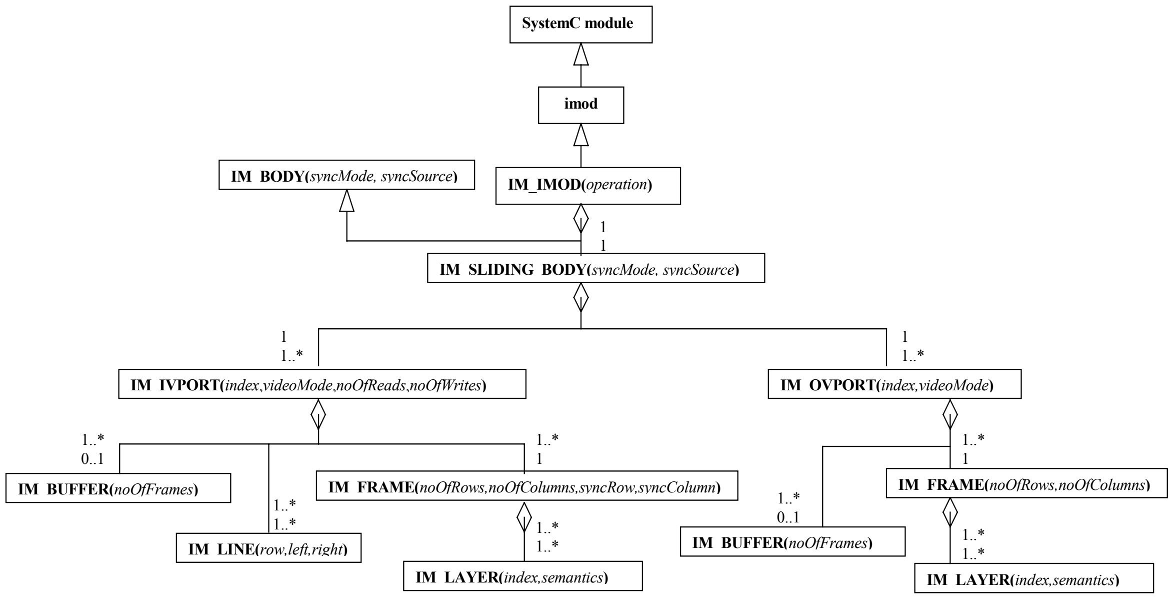
Figure 2. IMEM source-code.

3 IMEM module specification

An IMEM module specification typically specifies one video operation such as convolution, gray-scale transformation, segmentation or morphological operations such as open and close [5]. All these algorithms are neighbourhood oriented and can be divided into a spatio-temporal and a functional mapping. The functional mapping specifies how one output pixel is determined given a certain neighbourhood, that is, the description of the method that calculates the output pixel from a 3-dimensional collection of pixels. The spatio-temporal mapping, on the other hand, is a specification of that 3-dimensional neighbourhood and the spatial domain it is maintained on. An IMEM specification also specifies the input- and output video streams that interface the video processing algorithm within the same module.

3.1 Interface- and memory modelling using design entities

The IMEM UML class-diagram shown in Figure 3 shows how IMEM design entities can be combined, in order to describe interfaces and spatio-temporal mapping of a video processing algorithm.



SystemC module
imod
IM_BODY(syncMode, syncSource)
IM_IMOD(operation)
1
1
IM_SLIDING_BODY(syncMode, syncSource)
1
1..*
1
1..*
IM_IVPORT(index,videoMode,noOfReads,noOfWrites)
IM_OVPORT(index,videoMode)
1..*
0..1
1..*
1
IM_BUFFER(noOfFrames)
IM_FRAME(noOfRows,noOfColumns,syncRow,syncColumn)
1..*
1..*
IM_LINE(row,left,right)
1..*
1..*
IM_LAYER(index,semantics)
1..*
1
IM_FRAME(noOfRows,noOfColumns)
1..*
0..1
IM_BUFFER(noOfFrames)
1..*
1..*
IM_LAYER(index,semantics)

Figure 3. UML class-diagram of IMEM modelling-constructs.

IM_SLIDING_BODY has two parameters, *syncMode* and *syncSource*. *syncMode* can be set to either FREE_RUN or SYNC, depending on if the body and the output streams are synchronised with any of the input streams. If set to SYNC, *syncSource* can have the stream index of any input stream. IM_SLIDING_BODY inherits some of its behaviour from IM_BODY. An IM_SLIDING_BODY is owned by an IM_IMOD(operation) template class, which is uniquely defined for each video processing operation. Imod is the template class derived from a SystemC module. IM_IVPORT corresponds to an input video stream and have the parameters *index, videoMode, noOfReads* and *noOfWrites. index* is the stream index that has to be unique. v*ideoMode* is the sequence that pixels appear at the input port: NON_INTERLACED, INTERLACED_ODD or INTERLACED_EVEN. INTERLACED_EVEN means that even rows appear first, then odd rows. *noOfReads* and *noOfWrites* are used to model a relative difference on the video stream data speed according to the equation:

$$\frac{\textit{input stream speed}}{\textit{output stream speed}} = \frac{\textit{noOfReads}}{\textit{noOfWrites}}$$

noOfReads and *noOfWrites* should be interpreted as: during the time an output stream produces *noOWrites* data elements, *noOfReads* data elements are read from the input stream. Output video streams must always have the same data speed, but any input stream can have a relative data speed difference with respect to the output streams. IM_OVPORT is the entity that corresponds to an output video stream. *number* and *videoMode* have the same semantics as for IM_IVPORT. IM_BUFFER is the entity that corresponds to buffering on output or input video streams. These buffers are needed when the pixel sequence on a stream is changed, for instance from NON_INTERLACED to INTERLACED_ODD, or any other combination. *noOfFrames* is the only parameter and must be set to either exact size in number of frames, or to GENERIC, which means that the IMEM system is allowed to determine the buffer size. The frame size of any input or output video stream is set by the entity IM_FRAME and the parameters *noOfRows, noOfColumns. syncRow* and *syncColumn* is the spatial position that the body enters at frame synchronisation. It is possible to model different colour space models, such as Red-Green-Blue, or Hue-Saturation-Intensity, by mapping a colour model and its components onto layers. IM_LAYER and its parameters *index* and *semantics,* are used to associate a layer index with a string representing the semantics of a colour component such as RED. The order, in which the layers are assigned to the IM_FRAME entity, corresponds directly to the order colour components appear on the stream port. The first IM_LAYER assigned to IM_FRAME is also the first colour component that appears on port at frame synchronisation.

A structure of IM_LINE entities is used to model a body. IM_LINE has three parameters, *row, left* and *right. Row* is the relative row-axis position. *Left* and *right* corresponds to the number of pixels to the left and to the right of the body centeriod. Figure 4 is divided into two parts a) a geometric graph of a body and b) the corresponding UML class-diagram. The geometric graph also shows two examples of how individual pixels are addressed. This body consists out of three slices, where one slice is a spatial collection of pixels. The first slice, the oldest in the temporal dimension, is modelled with three instances of IM_LINE, owned by IM_SLIDING_BODY. This slice is referenced as having the relative frame number 0. The latest slice in this example has the relative frame address 2 and this part of the IM_LINE structure is depicted rightmost in the UML class graph.

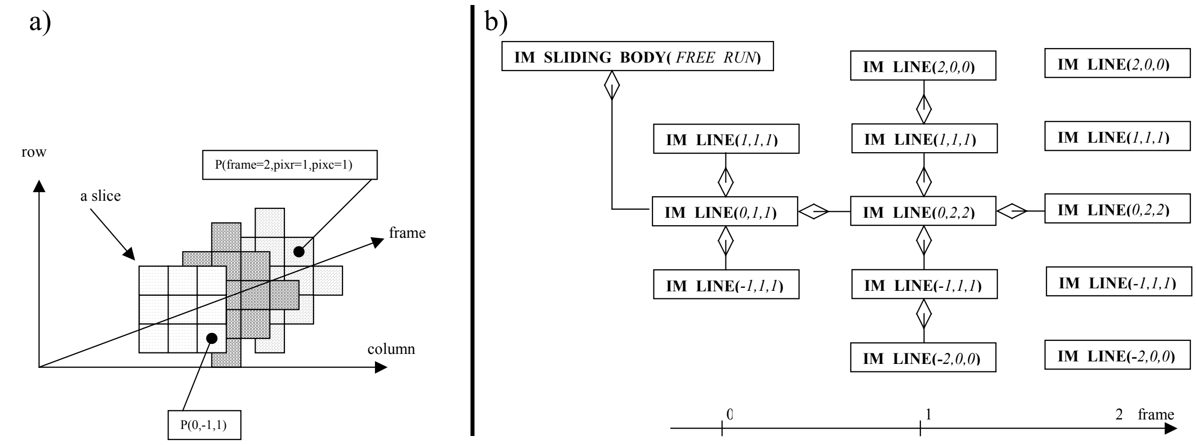


Figure 4. How to specify a body in IMEM.

3.2 Functional mapping

The functional mapping of a video processing algorithm, that is how the output pixel is determined from a 3-dimensional collection of pixels, is specified using a standard C++ programming style. All the relative pixel positions are supplied to the functional mapping through a method interface, *getPixelData(int _stream, int _slice, int _row, int _column, int _layer).*

3.3 Implementation of an IMEM model

Figure 5 shows how both the spatio-temporal and functional mapping can be specified in one single source code file. The IM_IMOD-directive at line 1 denotes the start of an IMEM specification, IM_EO_IMOD at line 72, denotes the end. IM_FUNCTIONAL at line 35, denotes the end of spatio-temporal mapping and the start of functional mapping. This simple mean-filter algorithm has only a five pixel spatial mapping, denoted at line 6-10. The double arrow operator << is used to assign a design entity to another. An instance of a design entity must have a unique identifier, such as *lnd* at line 6. The second pair of paranthesis encloses the design entity parameters. Line 3 instantiates an IM_SLIDING_BODY and assigns it at line 4 to the current instance of the mean-filter. Line 12-16 specifies the colour space model. Line 17 and 29 specifies input- and output streams. Input- and output frame format is specified at line 19 and 26. An IMEM specification is a template class with the data type carrying video data as a template parameter. This parameter is accessed with the VIDEO macro at line 37. SWITCH_STREAM, line 38, and SWITCH_LAYER, line 41 are macros used to select current output component at the functional mapping. Line 44-49 shows how the red colour component for output stream 1 is calculated as the mean value of a five pixel neighbourhood. The method *getPixelData(int _stream, int _slice, int _row, int _column, int _layer)* is called in order to access pixel positions relative to the current spatial body location.

```
IM_IMOD(mean)

  IM_SLIDING_BODY(sbody)(FREE_RUN);
  *this << sbody;
//                                                   This is a simple single slice body
  IM_LINE(lnd)(-1,0,0);            //                     __
  IM_LINE(lnu)(1,0,0);             //                 __ |__| __
  IM_LINE(bd0)(0,1,1);             //                |__| __ |__|
  bd0 << lnu;                      //                    |__|
  bd0 << lnd;                      //

  IM_LAYER(pix0)(0,"RED");         // Pixel format
  IM_LAYER(pix1)(1,"GREEN");
  pix0 << pix1;
  IM_LAYER(pix2)(2,"BLUE");
  pix1 << pix2;
  IM_IVPORT(iport1)(1,INTERLACED_EVEN,1,1);   // Input stream 1
  sbody << iport1;
  IM_FRAME(fri1)(119,199,0,0);
          fri1 << pix0;            // Assign pixel format
          iport1 << fri1;
          iport1 << bd0;           // Assign body
          IM_BUFFER(ibuf)(1);
          iport1 << ibuf;

  IM_FRAME(fro1)(119,199);         // Output frame format
  fro1 << pix0;                    // Assign pixel format

  IM_OVPORT(oport1)(1,NON_INTERLACED);       // Output stream 1
  sbody << oport1;
          IM_BUFFER(obuf1)(GENERIC);
          oport1 << obuf1;
          oport1 << fro1;

IM_FUNCTIONAL(mean)

  VIDEO vdt;
  SWITCH_STREAM
  {
  case 1:
          SWITCH_LAYER
          {
          case 0:
                    vdt = getPixelData(1, 0, 0, 0, 0);
                    vdt = vdt + getPixelData(1, 0, 1, 0, 0);
                    vdt = vdt + getPixelData(1, 0, -1, 0, 0);
                    vdt = vdt + getPixelData(1, 0, 0, 1, 0);
                    vdt = vdt + getPixelData(1, 0, 0, -1, 0);
                    vdt = (VIDEO)(vdt/5);                  // Mean value
                    return(vdt);
          case 1:
                    vdt = getPixelData(1, 0, 0, 0, 1);
                    vdt = vdt + getPixelData(1, 0, 1, 0, 1);
                    vdt = vdt + getPixelData(1, 0, -1, 0, 1);
                    vdt = vdt + getPixelData(1, 0, 0, 1, 1);
                    vdt = vdt + getPixelData(1, 0, 0, -1, 1);
                    vdt = (VIDEO)(vdt/5);                  // Mean value
                    return(vdt);
          case 2:
                    vdt = getPixelData(1, 0, 0, 0, 2);
                    vdt = vdt + getPixelData(1, 0, 1, 0, 2);
                    vdt = vdt + getPixelData(1, 0, -1, 0, 2);
                    vdt = vdt + getPixelData(1, 0, 0, 1, 2);
                    vdt = vdt + getPixelData(1, 0, 0, -1, 2);
                    vdt = (VIDEO)(vdt/5);                  // Mean value
                    return(vdt);
          }
          break;
  };
  return(0);

IM_EO_IMOD
```

Figure 5. IMEM-module source code.

4 A DESIGN EXAMPLE

A spatio-temporal noise reduction filter is captured using both IMEM and a standard SystemC modelling style. The simulation set-up, based on three different modules, is the same in both cases, see Figure 6. Istream (1), an input video streamer module, which serves as a part of the test-bench, providing the captured filter model with video data. Noise reduction filter (2), is the filter model and Ostream (3), is a part of the test-bench that will store filtered pixels onto disc. The output streamer object is also the only concurrent process, which will invoke the filter slave process through the Remote Procedure Call, provided in SystemC. The functionality of the filter slave process is alternatively captured using IMEM or SystemC standard style. Modelling- and simulation performance is then compared for the two cases.

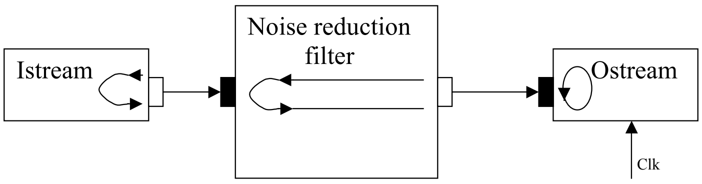


Figure 6. Simulation of the design example.

4.1 Filter description

The task of the noise reduction algorithm can be divided into two main sub-tasks: (A) to detect a part of the image and determine whether this is a part of a moving image, that is called local scene-change detection, (B) to filter out noise with local scene-change taken into account. A block diagram of the filter is depicted in Figure 7.

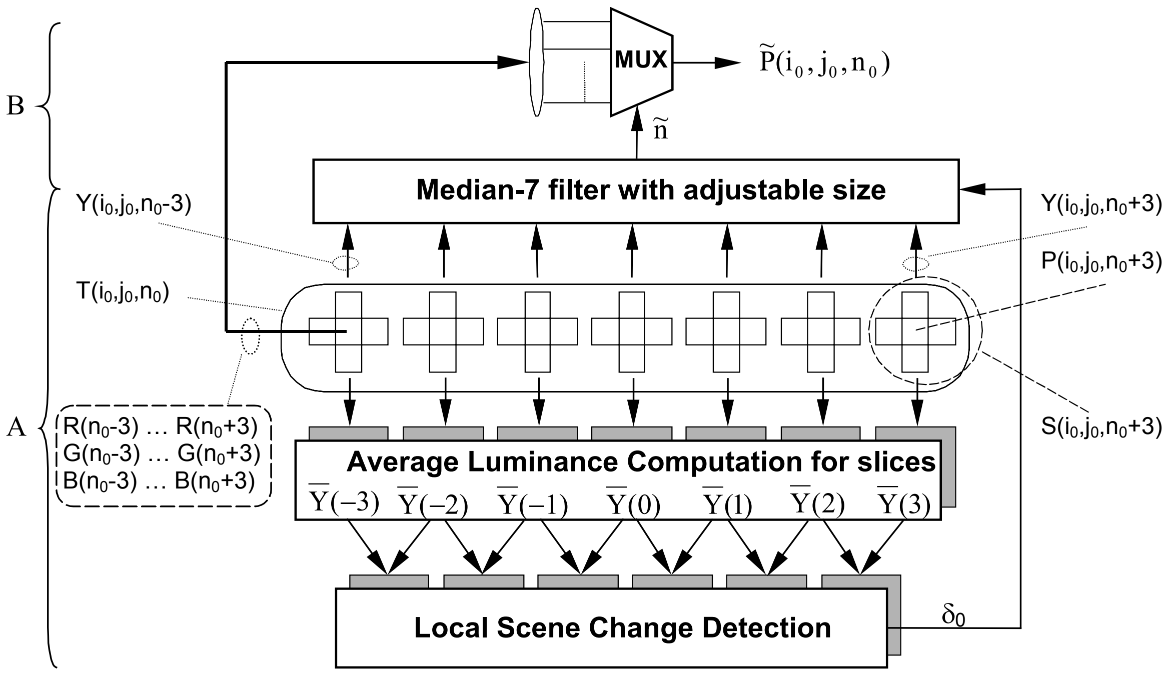

B
A
MUX
$\tilde{P}(i_0, j_0, n_0)$
$\tilde{n}$
Median-7 filter with adjustable size
$Y(i_0,j_0,n_0-3)$
$Y(i_0,j_0,n_0+3)$
$P(i_0,j_0,n_0+3)$
$T(i_0,j_0,n_0)$
$S(i_0,j_0,n_0+3)$
$R(n_0-3) \ldots R(n_0+3)$
$G(n_0-3) \ldots G(n_0+3)$
$B(n_0-3) \ldots B(n_0+3)$
Average Luminance Computation for slices
$\overline{Y}(-3)$ $\overline{Y}(-2)$ $\overline{Y}(-1)$ $\overline{Y}(0)$ $\overline{Y}(1)$ $\overline{Y}(2)$ $\overline{Y}(3)$
Local Scene Change Detection
δ_0

Figure 7. Block diagram for the spatio-temporal video filter.

The notation and definition used in the algorithm description are: A frame *F(n),*

$$F(n) = \{R(n), G(n), B(n)\},$$

is a matrix of RGB-values (colour components) in the n:th frame. A pixel *P(i, j, n),*

$$P(i, j, n) = \{R(i, j, n), G(i, j, n), B(i, j, n)\} \in F(n),$$

is an element in *F(n)* with the spatial position *(i, j)*. A slice,

$$S(i_0, j_0, n_0) \subset F(n_0),$$

positioned at *(i_0, j_0)* in the n:th frame includes the pixel *$p_0(i_0,j_0,n)$* and a portion of the n:th frame that surrounds the pixel p_0. A tube,

$$T(i_0, j_0, n) = \{S(i_0, j_0, n) \mid (n_0 - 3) \le n \le (n_0 + 3)\},$$

is a set of slices with same *(i, j)* but located in consecutive frames.

4.2 Algorithm

This section outlines the behaviour of the filter algorithm. The first step of the algorithm is to calculate the average luminance for each slice in a tube.

$$\overline{Y}(n)\Big|_{n_0 - d \le n \le n_0 + d} = \frac{1}{5} \cdot \left(\sum_{[i,j] \in S(i_0, j_0, n)} \text{luminance}\{P(i, j, n)\} \right)$$

Using the average luminance for a slice and the blue colour component between two in time adjacent pixels, the differences between these two pixels are calculated. If either of the differences is higher than a certain threshold level (T_y and T_b), a scene change is indicated in a vector, *I*.

$$I(n, n-1) = \begin{cases} 1 & \text{if} \quad |Y(n) - Y(n-1)| > T_y \vee |B(i_0, j_0, n) - B(i_0, j_0, n-1)| > T_b \\ 0 & \textit{otherwise} \end{cases}$$

From the scene change vector, *I*, the length, δ_0, from a scene change to the centre pixel determines the length used by the median filter. The luminance from the centre pixel in the tube and the median filter width, δ_0, are the inputs to the median filter.

$$Y(i_0, j_0, \tilde{n}) = Median\{Y(i_0, j_0, n)\Big|_{n_0 - \delta_0 \le n \le n_0 + \delta_0}\}$$

The filter output is selected from the centre pixel's original RGB-values in frame number ñ.

4.3 IMEM model of the filter

The body model diagram, depicted in Figure 8, shows a collection of pixels that represents the spatio-temporal mapping of the noise reduction filter. Seven consecutive slices, each one of them consisting out of five adjacent pixels, form a three dimensional body. The object-relation diagram in Figure 9 shows how the design entities in IMEM are used to capture the neighbourhood and the stream port interfaces. 21 objects of the IM_LINE entity, connected in a structure, represents the neighbourhood depicted in Figure 8.

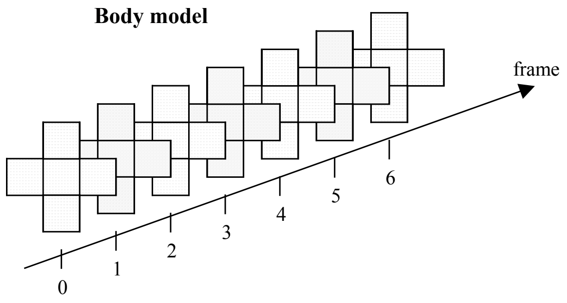


Figure 8. The 3-dimensional neighbourhood of pixels that the filter operates on.

Both input and output frame sizes are 576 x 720 pixels, as being set by the IM_FRAME objects. Colour components are mapped onto layer one, two and three, which is the same order they appear on the video streams. The input stream has one buffer with the size of a single frame assigned to it. The output stream is not synchronized with the input stream. This is defined by the IM_SLIDING_BODY object and the parameter value FREE_RUN. The functional mapping of this adaptive median filter is mainly specified within a separate standard C++ class. This reusable software component is then equally invoked within both the IMEM specification as well as the standard SystemC Master-Slave module.

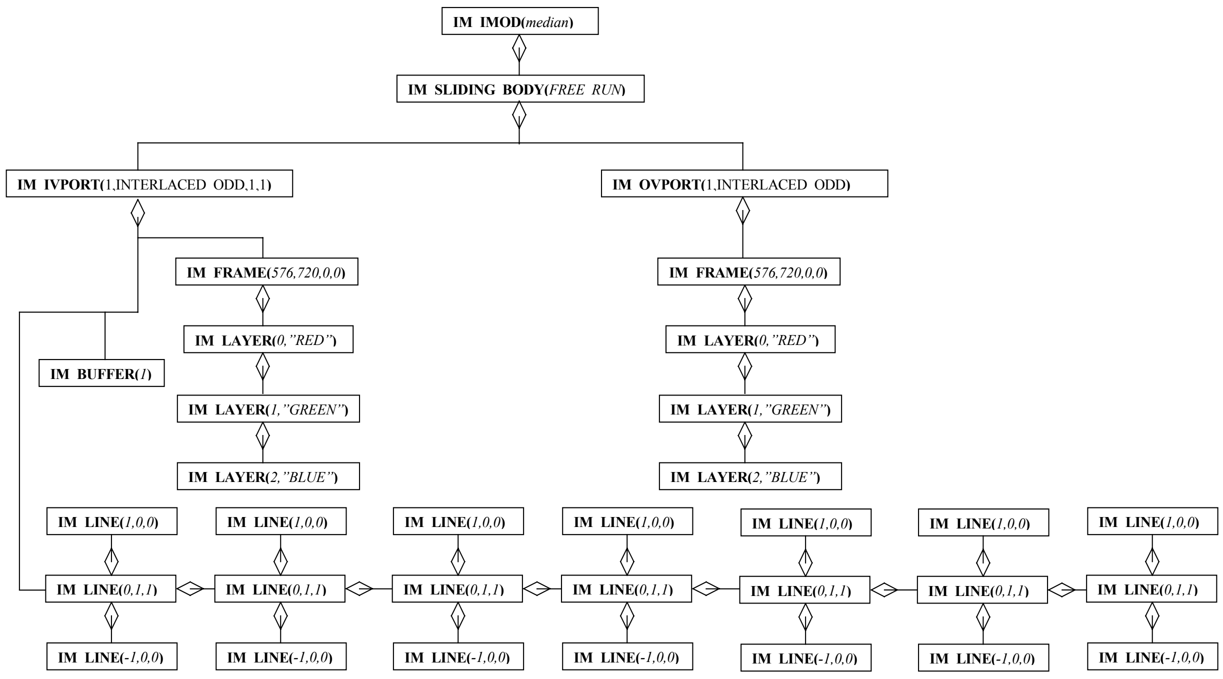


Figure 9. IMEM object-relation diagram for the noise reduction filter.

5 RESULT

This section presents the results from the comparison between IMEM and SystemC Master-Slave Communication Library (version 2.0 Beta-3) using the video filter design as test case. The comparison was made on a Pentium III (800MHz) computer running Windows 2000. We have used the number of source code lines as a measure of the modelling efficiency, see Table 1. The IMEM model is separated on three code blocks, (1) the *sc_main* routine, (2) the filter kernel and (3), the filter component description. The standard SystemC Un-Timed Functional model is captured on four different code blocks, (1) the *sc_main* routine, (2) the frame memory model, (3) the filter kernel and (4), the filter component description. The filter kernel, captured as a standard C++ class is the same for both models. The *sc_main* routine only differs at the kind of filter model that was invoked, IMEM or standard SystemC. The frame memory module that was only used for the standard SystemC model is a generic frame memory model that can be packaged into a library for later reuse. For that reason, the most realistic numbers to compare is the line representing the two different component descriptions, 145 versus 264 lines.

PARAMETER	IMEM	SYSTEMC UTF MODELLING
Simulation speed	13.1 seconds/frame	10.6 seconds/frame
Lines of code		
frame memory module		652
main	115	115
filter kernel module	290	290
filter component description	145	264
Total number of lines	550	1321
Extraction of data dependency	Built into the model	Simulation and data profiling

Table 1. Comparison of modelling- and simulation performance.

The numbers in Table 1 clearly indicates that IMEM as a specification method, reduces the number of source code lines needed to capture our spatio-temporal noise reduction filter by 45%, but to the cost of an 23% increase of simulation time. The simulation mechanism in IMEM is generalised for any neighbourhood oriented video processing operation which interfaces with any number of input and output video-streams. We have selected an object-oriented and highly modular implementation to make IMEM general. The use of modularity, polymorphism and virtual methods explains the simulation performance degradation. As indicated on the last row of Table 1, the IMEM filter module specification has an explicit built-in data dependency model. Extracting the same data dependency from the standard SystemC model would require some additional profiling.

6 CONCLUSION

In this paper, we have shown that a 3-dimensional collection of pixels is a natural and modelling efficient abstraction for video processing operations. To support this abstraction, an object-oriented specification method, IMEM was presented. This method was evaluated and compared with a SystemC Un-Timed Functional specification, exposed on a realistic test-vehicle. IMEM was found to be most modelling efficient in terms of number of source code lines, although to the cost of a decrease in simulation speed. The benefits of using IMEM would probably have been even greater if the test-vehicle would have been more complex and if the designer's modelling- and debugging time would have been taken into account. What is even more important, is that the structure of design entities, being a conceptual memory and interface model, explicitly reveals data dependency information, which is important input to high-level synthesis, rapid prototyping, data reuse and loop transformation analysis. These are areas of interesting and challenging research that we will address in future. The data dependency modelling and the higher abstraction in IMEM will serve as an excellent extension to the already well known SystemC workflow. Adding more modelling- and analysis capabilities to this single environment workflow, will definitely help video systems designers to reduce the time to market.

Acknowledgements

The Mid-Sweden University in Sundsvall, Sweden and the KK-Foundation, Sweden are gratefully acknowledged for their financial support.

References

[1] S.J. Hill, D. Crookes and A. Bouridane, "The use of high level tools for developing volume graphic and video sequence processing applications", *Proceedings of 7th international congress on image processing and its applications, IEE 1999, (Conf. Publ. No.465).*

[2] *SystemC User's Guide*, Version 2.0, http://www.systemc.org

[3] *Master-Slave Communication library User's Guide,* Version 2.0 Beta-3, http://www.systemc.org

[4] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.L. van Meerbergen and A. van der Werf, "Modelling Periodicity by PHIDEO Streams", *Proceedings of 6th International Workshop on High-Level Synthesis, pp. 256-266, 1992.*

[5] *Digital Image Processing*, R.C. Gonzales and R.E. Woods, Addison Wesley 1993.

[6] S. Wuytack, J.Ph. Diguet, F.Catthoor and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.6, no.4, 1998.*

[7] L. Nachtergaele, F. Catthoor, F.Balasa, F. Franssen, E. De Greef, H. Samsom and H. De Man, "Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems", *Records of the 1995 IEEE international workshop on memory technology, design and testing, 1995.*

[8] Dace C.A. "An applicative high-level language for dsp system design", *IEE Colloquium on General-Purpose Signal-Processing Devices (Digest No.085) 1993*

[9] Kazimierz Wiatr, "Dedicated hardware processors for real-time image data pre-processing implemented in FPGA structure", *Proceedings of ICIAP 97. 9th International Conference on Image Analysis and Processing, vol.2, pp 69-76.*

[10] Luis L. Nozal, Gerardo Aranguren, José Luis Martín and Joseba Ezquerra, "Moving images time gradient implementation using RAM-based FPGA", *Proceedings of the SPIE - The International Society for Optical Engineering 1997, vol.3028, pp.108-116.*

[11] Brad Taylor, "DSP filters in FPGAs for image processing applications", *Proceedings of the SPIE - The International Society for Optical Engineering 1996, vol.2914, pp.100-109.*

[12] B. Oelmann, H. Norell, R. Andersson, Y. Xu, "Design of Real-Time Signal Processing ASIC for Noise Reduction in Moving Video Images", *Proceeding of IEEE Norchip Conference 1999, pp.228-33.*

[13] B. Thörnberg and M. O´Nils, "Analysis of modeling and simulation capabilities in SystemC and Ocapi using a video filter design", *Proceeding of ECSI forum on design languages 2001.*