Abstract—Fully configurable video coding is a novel approach to video compression in which a video decoder can be dynamically constructed based on a decoder description. This paper presents a new syntax for describing video decoder functionality and structure, namely, the Decoder Description Syntax (DDS). The DDS is a platform-independent syntax that can define all aspects of a video decoder in terms of basic processing instructions. A DDS description of a video decoder may be coded and transmitted, and then executed by a generic processing platform, a “Universal Video Decoder”. Any new or modified video decoding function may be described, communicated and instantiated using the DDS, which makes it possible to rapidly implement new coding algorithms, to dynamically adapt the coding algorithm to suit the video data, and to efficiently implement multiple coding formats on one platform. We present examples of video decoding functions implemented in the DDS and show how these may be executed on a Universal Video Decoder. We demonstrate that flexible configuration, reconfiguration, and decoding of video can be achieved using a real-time prototype Universal Video Decoder with functions transmitted and instantiated on-the-fly using the DDS.

Index Terms—video coding, reconfiguration, syntax

I. INTRODUCTION

Current video communication systems are based on video coding standards such as MPEG-2, VC-1 and H.264/AVC [1]. These standards each define a compressed video format and a method of decoding this format, enabling interoperability between encoder and decoder implementations. However, increasingly, device manufacturers must support multiple, incompatible coding standards [2] leading to over-designed decoding products. Bringing a new video coding concept to market is a long, expensive process. Due to the need for selecting appropriate combinations of technologies in each new generation of standards, many research innovations never make it into consumer products. In contrast, many of the tools specified in a standard are never used in particular applications. For example, the ATSC Mobile/Handheld Digital TV standard [3] specifies a reduced subset of the H.264/AVC Baseline Profile in recognition that certain options are not required for this application.

Furthermore, a fixed, standardized video coding algorithm will not be optimal for all types of video content and application scenarios. As one example, consider content that varies between filmed scenes and animation. It is likely that a system using a fixed codec such as MPEG-2 or H.264/AVC could be improved upon in terms of rate-distortion (RD) performance. Specific functions could be tailored to compress the animation sections of the sequence and transmitted to the decoder. With the increasing diversity in applications for generating, transmitting and viewing digital video, there is a corresponding diversity in the nature and statistical properties of video sources. In many applications these will include high-quality “natural” video, video captured from poor-quality image sensors, animation, graphic overlay, video that has already been heavily compressed, gaming scenes, composite scenes, etc. Each type of source has distinct statistical properties that could benefit from tailored, specific modifications to the video compression algorithm.

Additionally, it is well known that better RD performance may be obtained by adaptive coding techniques in which the video compression algorithm is changed to suit the particular characteristics of the video content [4,5,6,7]. Recent standards such as H.264/AVC have attempted to improve adaptivity by specifying a wide range of coding options or tools. Choosing the best coding option based on bitrate and image quality via rate-distortion optimized mode selection gives improved compression performance [8]. However, the performance gains achieved through this approach are limited by the predefined options specified in the standard.

It is possible to avoid many of these obstacles through a new approach to defining and implementing video decoders. Ideally, a solution is desired that enables flexible reconfiguration for efficient decoding of multiple video formats, rapid deployment of new coding algorithms, and dynamic adaptation to suit the video content. This requires a method of defining video decoders that:

- can fully describe a video decoder’s functionality and structure;
- is platform independent;
- is capable of being dynamically executed (implemented) on a variety of processing platforms;
- is capable of describing existing and new formats for coded video; and,
- can be modified dynamically during a video communication session.

Building on earlier work [9], MPEG’s Reconfigurable Video Coding (RVC) initiative has developed two standards that address these requirements. Indeed, the initiative permits new technologies to be adapted into existing standards. In the RVC model, a decoder is specified as an interconnected set of Functional Units (FUs) [10], which are decoding tools such as inverse transforms and entropy decoders. The available FUs are specified in the Video Tool Library (VTL) standard [11]. A particular decoder is defined by a Codec Configuration Representation [12], which describes a bit stream format and a set of FU interconnections and parameters. The Codec Configuration Representation is specified prior to starting a decoding session. Hence, an RVC video decoder is constructed from a set of pre-defined decoding tools. An existing or new video format can be supported by reconfiguring the RVC decoder, provided the format uses standard FUs from the Video Tool Library. Two key constraints deliberately selected for the RVC model are:

1. A coding format must use combinations of coding tools (FUs) that are specified in the VTL. Introducing a new tool that is not in the VTL requires a lengthy process of standardizing and disseminating a new version of the VTL.

2. Dynamic re-configuration is not currently supported, i.e., the RVC model does not support fully adaptive coding in which the coding algorithm adapts to suit the time-varying characteristics of the video sequence.

This RVC approach goes a long way toward addressing the obstacles presented by a standardized video decoder, and we note that the above constraints will provide for efficient decoder implementations in many scenarios. However, we argue that these constraints are not always necessary, rather that, through the use of a new syntax for describing and configuring a video decoder, they can be selectively employed to achieve rate-distortion-complexity gains. We propose a framework centered on a Decoder Description Syntax (DDS) that may be transmitted and implemented on a “universal” video decoder in order to create and modify video decoding function and structure in real time; Figure 1 illustrates the proposed framework.

![Diagram](image)

**Figure 1**: Configurable video coding using a decoder description syntax.

In the following sections we give an overview of this new syntax and demonstrate its usage in defining and implementing configurable video coding structures. We note that while emphasis is placed on the features of the framework that permit dynamic reconfiguration of codecs, this syntax is also appropriate for use with RVC even with the aforementioned constraints. In particular, the RVC library of Functional Units could be available at the coder and decoder as a baseline for reconfiguration, and the DDS simply used to encode and transport necessary FU selections and interconnections.

II. OVERVIEW OF THE DECODER DESCRIPTION SYNTAX

In this work we present a Decoder Description Syntax (DDS) that has two important advantages over existing approaches. First, the proposed DDS captures video decoder functionality and structure using low-level, platform-independent primitives. This makes it possible to directly describe any decoding process using the DDS, without the need for a standardized library of video decoding tools. Second, the DDS is capable of dynamic execution and reconfiguration. A decoding process can be configured and reconfigured in real time during a coding session, making it possible to adapt the decoding algorithm to suit the data.

The DDS is a platform-independent syntax designed to communicate the functionality of the video decoder. The configuration decoder at the receiver detects and decodes the DDS from the bit stream and constructs, that is, instantiates, the signaled video decoder functions. The DDS can be transmitted within the compressed video bit stream or as side information, depending on the application scenario, and may be used to signal a completely new decoder, new or changed functionality, and corresponding changes to coded video bit stream parsing functions. A platform-independent configuration syntax is necessary to avoid the necessity of specifying a separate set of configuration instructions for each possible decoding platform. We believe that a platform-independent DDS can offer significant value by capturing the configuration required to decode a particular video format in a single description.

Figure 2 shows a model of potential DDS incorporation in a configurable coding scenario. A decoder design is defined by the encoder and specified in a high-level language. This is converted into the Decoder Description Syntax which provides a complete structural and functional definition of the decoder. The DDS is transmitted, for example, as part of the coded video bit stream, and decoded or interpreted. A generic Universal Video Decoder (UVD) implements the decoder design by interpreting the DDS and executing appropriate processes. As an example, the video encoder may adapt its coding algorithm to improve rate-distortion performance. The decoding algorithm is modified accordingly and this is signaled by sending new or updated DDS. The UVD implements the modified DDS and continues to decode video using the new algorithm.
The proposed Decoder Description Syntax differs from the MPEG RVC approach in the following ways. In RVC a video decoder is constructed as a network of interconnected functional units. The topology of this network is specified using an XML-based Decoder Description Language (DDL). RVC’s DDL does not provide for describing the internal processing of the functional units. Additionally, RVC separates the parsing process from the decoding tools (or FUs) to handle two different types of data. The parsing process is handled separately by specifying the structure of the bit stream using a Bit-stream Syntax Description Language (BSDL). In contrast, the proposed DDS is able to describe the entire functionality of the functional units and their interconnections using platform-independent functional processing primitives. Furthermore, the DDS does not differentiate between the parsing process and the rest of the decoding process of a video decoder. Since all the parsing functions are entirely described using DDS, the coded video bit stream need not be specified separately. Therefore, the DDS is designed to be able to describe the entire decoding process. Furthermore, it was designed to limit reconfiguration overhead as much as possible.

### III. THE DECODER DESCRIPTION SYNTAX IN DETAIL

Once a new or modified function is determined by the high-level decoder design block illustrated in Figure 2, the source code must be encoded, transported to the receiver, and instantiated at the decoder. The DDS can be broadly divided into two main components that handle the first two operations:

1. Functional Processing Primitives – elements and structures needed to describe the functional processes that make up a video decoder; and,

2. Transmission Syntax – encapsulates the Functional Processing Primitives to ensure that they are suitable for efficient encoding and transmission or storage.

The following two sections describe these two components in detail. However, due to space constraints, only selected examples of the functional processing primitives and transmission syntax elements are shown. A complete description is available at [14].

#### A. Functional Processing Primitives

The video decoder is divided into a number of processing units or functions. The internal processing of these units and the connections between them, that is, function calls, determine the functionality of the video decoder. All decoder functions are required to be self-contained, that is, without global dependencies, as this allows for platform-specific implementations such as parallelization and pipelining at the receiver end. The functional processing primitives are used to describe the internal processing of these functions and their interconnections.

Each function is divided into data and program segments. The simple example given in Figure 3 shows the main segments of a function, the data declarations and the program instructions, which include function calls. These segments will be described in the following sections. We note that these segments for this prototype DDS were designed for efficiency in transmission of video codec reconfiguration. The data and instruction types for the Program Segment could certainly be expanded or reduced depending on the implementation. Furthermore, the instruction format, opcode representation, could be modified and implemented using alternative tools such as the Low Level Virtual Machine [13].

```c
return type FunctionName(input parameters) {
    Variables
    Constants
    Instructions
    ... return parameter
}
```

![Figure 3: Components of a DDS function.](image)

#### 1) Data Segment

The data segment includes all the variables and constants declared in the function, including the input and output parameters. Table 1 shows the data types that are currently supported by the DDS.

<table>
<thead>
<tr>
<th>Type name</th>
<th>Type_ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U_CHAR_V</td>
<td>0</td>
<td>Unsigned character (1 Byte)</td>
</tr>
<tr>
<td>CHAR_V</td>
<td>1</td>
<td>Signed character (1 Byte)</td>
</tr>
<tr>
<td>U_INT_V</td>
<td>2</td>
<td>Unsigned integer (4 Bytes)</td>
</tr>
<tr>
<td>INT_V</td>
<td>3</td>
<td>Integer (4 Bytes)</td>
</tr>
<tr>
<td>DOUB_V</td>
<td>4</td>
<td>Fractional number (4Bytes integer, 4 Byte decimal part in BCD)</td>
</tr>
<tr>
<td>BOOL_V</td>
<td>5</td>
<td>Boolean (1 Bit)</td>
</tr>
<tr>
<td>U_CHAR_P</td>
<td>6</td>
<td>unsigned character pointer (array)</td>
</tr>
<tr>
<td>CHAR_P</td>
<td>7</td>
<td>character pointer (array)</td>
</tr>
<tr>
<td>U_INT_P</td>
<td>8</td>
<td>unsigned integer pointer (array)</td>
</tr>
<tr>
<td>INT_P</td>
<td>9</td>
<td>integer pointer (array)</td>
</tr>
<tr>
<td>DOUB_P</td>
<td>10</td>
<td>double precision floating point pointer (array)</td>
</tr>
<tr>
<td>BOOL_P</td>
<td>11</td>
<td>Boolean pointer (array)</td>
</tr>
</tbody>
</table>
Note that the pointer types do not have specified storage or precision requirements because this depends on the underlying hardware platform of the receiver. The above types have been chosen as a compromise between flexibility and transmission efficiency. For example, consider a structure of a two dimensional array or a matrix to be included as a basic type. This new structure could be typed for all six basic types and these new structures then supported with a new set of matrix instructions to accesses the data. Conversely, the matrix structure can easily be implemented by expanding 2-D or higher order addressing into 1-D address space and using the pointer or array types without any additional memory requirement.

Table 2: Instruction types.

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic operations</td>
<td>Arithmetic, logical, bitwise operations, flow control, conditional statements, etc.</td>
<td>Required to construct basic program logic and functionality.</td>
</tr>
<tr>
<td>Advanced operations</td>
<td>Memory management, File read / write.</td>
<td>Specialized tasks such as memory and file management.</td>
</tr>
<tr>
<td>Video coding specific operations</td>
<td>Bitstream handling, array processing, data movement.</td>
<td>Common primitive tasks for video decoding.</td>
</tr>
</tbody>
</table>

2) Program Segment

The program segment consists of the set of function processing instructions. The instructions supported by the proposed DDS fall into three categories: basic, advanced and video coding specific; see Table 2. The set of instructions is designed as a compromise between flexibility and performance, with the aim of supporting a wide range of processing functions without sacrificing computational efficiency or expanding overhead significantly. Basic and advanced operations are designed to be capable of specifying any arbitrary functional process of the video decoder. Video Coding Specific (VCS) operations are designed to streamline the handling of processor-intensive tasks commonly carried out in a video decoder, such as bit stream handling, array processing, data re-ordering and data movement. A list of instructions that is a reduced, yet sufficiently complete, set of structured language instructions, which are currently supported by the DDS can be found at [14].

Bit stream handling functions are an important subset of VCS operations. Generally, almost all compressed video streams consist of variable-bit-length, non-byte-aligned coded elements. Typically, a parser in the video decoder reads these elements from a bit/byte stream. Therefore, native handling of these, usually computationally-intensive operations, is essential. A number of bit stream handling instructions are introduced in the DDS to facilitate efficient parsing functions. A selection of these is listed below in C-language syntax.

- unsigned int ReadBits(unsigned char count) - This instruction reads a specified count number of bits from the current point in the designated bit stream, typically an input coded video stream, and returns the value as an unsigned integer. This instruction can read up to a maximum of 32 bits from the bit stream.
- unsigned int PeekBits(unsigned char count) – This instruction is similar to ReadBits, however, the bit stream pointer is not incremented.
- bool ReadFlag(void) – The instruction reads one bit from the bit stream and returns “true” if the bit is set.

All instructions conform to the Standard Instruction Syntax (SIS) shown in the left column of Table 3. The SIS consists of seven byte fields starting from the OPCODE which identifies the instruction. It was observed that most commonly used instructions, that is, arithmetic, logical, and bitwise operations, consist of an output operand and two input operands in the following form:

\[(\text{type}) \text{out} = (\text{type}) \text{in1} \langle op \rangle (\text{type}) \text{in2} \quad (1)\]

where \(\langle op \rangle \in \{+, -, \text{!}, \text{!}, \text{>,}, \ldots\}\) and “type” specifies the enumerated types of the output operand \(\text{out}\) and the two input operands, \(\text{in1}\) and \(\text{in2}\), which are given in Table 1. Therefore, the SIS was designed with fields required to specify the type and the index of these three operands in order to constrain reconfiguration overhead. In the DDS all data elements of a function are identified through indexed lists for each data type.

Table 3: Standard Instruction Syntax (SIS).

<table>
<thead>
<tr>
<th>Field</th>
<th>Description in relation to basic operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td>Identifies the instruction or operation</td>
</tr>
<tr>
<td>TYPE_O</td>
<td>Type of output</td>
</tr>
<tr>
<td>OUT</td>
<td>Index of output</td>
</tr>
<tr>
<td>TYPE_I1</td>
<td>Type of input 1</td>
</tr>
<tr>
<td>IN1</td>
<td>Index of input 1</td>
</tr>
<tr>
<td>TYPE_I2</td>
<td>Type of input 2</td>
</tr>
<tr>
<td>IN2</td>
<td>Index of input 2</td>
</tr>
</tbody>
</table>

Simple instructions such as assignment operations may require only one input and one output operand and therefore the rest of the fields are ignored. More complex instructions may require other information to be carried by SIS fields and may occupy more than one instruction space. Below is an example of how a control flow “while loop” is specified using the SIS. The generic template and the program structure of a while loop are shown in Figure 4. The loop is specified using the loop declaration instruction followed by condition evaluation and loop processing instructions. The corresponding SIS entry of the loop declaration instruction is shown in Table 4. Note that all instructions permitted in the
current DDS prototype, ranging from arithmetic to flow control, are implemented in an analogous manner to the \texttt{while} instruction; further details can be found at [14].

\begin{figure}[h]
\centering
\begin{tabular}{c|c}
\hline
\textbf{Field} & \textbf{Value} \\
\hline
OPCDE & 35 \\
\hline
TYPE_O & \texttt{condition end offset} (N) This is the offset in instructions between current instruction position of this entry and the last instruction in \texttt{condition} section. This offset value should be less than 255. If more, TYPE_O should be set to 255 and OUT field is used for obtaining the actual value. \\
\hline
OUT & \texttt{index of condition end offset} The type of OUT is assumed to be integer. If TYPE_O =255, an index value is supplied here to get the actual offset value from an integer variable. (NOTE: these rules apply to all offsets and alternative indexes) \\
\hline
IN1 & \texttt{index of condition result} The type is assumed to be Boolean. If a Boolean variable is used instead of a condition evaluation, the index of the Boolean variable should be here. If condition evaluation instructions are present, then the result should be stored in a temporary Boolean variable and the index should be provided here. \\
\hline
IN2 & \texttt{loop end offset} (N + M) The offset from the current loop declaration instruction and end of loop processing instructions. The prior offset rules apply. \\
\hline
\end{tabular}
\caption{The \texttt{while} loop declaration instruction.}
\end{figure}

3) Function Calls

Functions are identified using a unique function identifier, $F_n.ID$, and each consists of its own data and program sections. Calls to other functions are implemented by using the SIS function call instruction by supplying the $F_n.ID$ of the called function. Since global data variables are not supported, in order to preserve independence between functions, all input parameters are copied to the called function. The processing results are also copied back to the calling function through return and input parameters.

Application-level communication is implemented using a mandatory standard root function with reserved function identifiers, $F_n.ID=0$. The execution of the video decoder is triggered by the application layer through this function which conforms to the following prototype expressed in C syntax:

\begin{verbatim}
void $F_0$(unsigned char * in_data, unsigned char * out_data, int &size); 
\end{verbatim}

In this function prototype, \texttt{in_data} is the byte buffer containing the input coded video bit stream and \texttt{out_data} is the byte buffer containing the decoded video data. However, the internal processing of this function is left completely unspecified and therefore must be signalled through the DDS. Video decoder execution is typically triggered by placing coded video data in \texttt{in_data} and indicating the amount of data using \texttt{size}. The video decoder specified by the DDS can now decode the compressed video data and is responsible for copying the result which is typically a decoded image or block back to \texttt{out_data}. The number of bytes in the result is indicated using \texttt{size}.

4) Public data

Video decoders typically use data structures to handle publicly available components such as, decoded picture buffers, sequence/picture parameters and Entropy decoding tables. These data structures are also handled in the same way as functions. Although the functions are self contained, the data and program segments are publicly accessible. Therefore, a decoded picture can be implemented using a function with arrays for pixel data. Entropy coding (e.g. VLD) tables can also be treated in a similar manner. If a new table must be created or changed, these can be transmitted as function parameters as described below.

B. Transmission Syntax

A transmission syntax is defined to encapsulate processing primitives for efficient transmission or storage. The configuration engine at the receiver, the UVD, decodes the transmission syntax, retrieves the functional processing primitives and constructs the signaled video decoder. The DDS information is added to the video bit stream or otherwise signaled to the UVD whenever a configuration change at the decoder is required. Figure 5 shows an overview of the transmission syntax in which the actual syntax elements are written in bold type. The DDS information in the bit stream is identified by DDS_Header which is followed by a DDS_Command that signals the type of configuration information to follow. Currently, the DDS_Command can indicate creation of new functions, modifications to existing functions, or deletion of functions, and includes a delete-all command to completely destroy the existing video decoder. Abstract level syntax for \texttt{New_Function} and \texttt{Modify_Function} is shown in Figure 6 and Figure 7, respectively.
case CREATE_FNS:
   No_Of_Fns_Minus_One
   for (i=0; i<(No_Of_Fns_minus_one + 1); i++)
   {
      New_Function()
   }
   exit

case MODIFY_FNS:
   No_Of_Fns_Minus_One
   for (i=0; i<(No_Of_Fns_minus_one + 1); i++)
   {
      Modify_Function()
   }
   exit

case DELETE_FNS:
   No_Of_Fns_Minus_One
   for (i=0; i<(No_Of_Fns_minus_one + 1); i++)
   {
      Fn_ID
   }
   exit

case DELETE_ALL:
   exit

Figure 5: Overview of the transmission syntax.

<table>
<thead>
<tr>
<th>Fn_ID</th>
<th>Prog_Size</th>
<th>U_CHAR_V_Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For (i=0; i < Prog_Size; i++)
{
   Instruction
}
if (U_CHAR_V_Size > 0)
{
   No_Of_Init_Values
   for (i=0, i<No_Of_Init_Values, i++)
   {
      Index
      Value
   }
   if (U_CHAR_P_Size > 0)
   {
      No_Of_Init_Pointers
      for (i=0; i<No_Of_Init_Pointers; i++)
      {
         Pointer_Index
         Size_In_Own_Type
         Value_Count
         for (j=0, j<Value_Count, j++)
         {
            Value_Index
            Value
         }
      }
   }
}

Figure 6: Sample syntax: instructions and two variable types in a new function.

Each new function is identified using a Fn_ID. If the Fn_ID of a new function matches that of an existing function, then the existing function is replaced with the new one. The first part of the syntax for a new function indicates the number of instructions present and sizes of variables or constants used in the program segment. Each variable or constant is identified using an indexed list for each type. The latter part of the syntax includes processing instructions. Typically, a function may contain constant values, arrays with pre-defined sizes and initial values. These are signaled for the data types present in the current function. Due to space limitations Figure 6 only shows the syntax for unsigned character variables and pointers.

Figure 7 shows sample syntax for modifying existing functions. Currently supported modifications are: data deletion for removing variables or constants; data insertion for introducing new variables or constants; and, delete program and insert program for removing or adding new instructions. Typically data and program changes are interdependent and therefore a flag is used to indicate additional modifications without moving up in the syntax hierarchy.

IV. IMPLEMENTATION EXAMPLES

This section presents a practical implementation of the DDS within a configurable video coding framework. First, the implementation framework of the universal video decoder engine is explained. The process of describing and transmitting the decoding functionality is explained using a single, illustrative decoding function. A real-time software prototype is developed to demonstrate dynamic reconfiguration of the
decoding process using the DDS. An encoder signals the DDS of a basic video decoder, consisting of a number of decoding functions such as entropy decoding, inverse quantization and inverse transform. Then the inverse transform function is replaced on-the-fly with a new transform during the decoding process. Screen shots and performance results of the software prototype are also presented.

A. Implementation framework

Currently the FCVC framework is targeted for software based implementation. The implementation framework of the universal video decoder engine is designed so that the UVD can be embedded into a typical video coding application. Figure 8 shows a simplified representation of the implementation framework. The interface with the application environment is through two buffers for input and output. The input from the application environment is typically a bit stream of data, which can be coded video or configuration data. The output is decoded video frames. For example, in a file based application, the input buffer will be filled up by the application layer by reading in from a file. In a real time application, e.g. using Microsoft Direct Show filters, the input will be the coded video frame from the upstream filter. Note that the application layer need not be aware of the composition of the input data itself. The universal video decoder identifies any reconfiguration information through a DDS identification marker embedded into the bit stream. Any reconfiguration DDS present in the bit stream is fed into the reconfiguration engine which then creates or modifies the video decoder. The coded video present in the bit stream is directly passed on to the video decoder through the standard root function (see section III B) of the decoder. The output of the video decoder is transferred to the video display buffer.

The current implementation specifies the operation and interface requirements between the UVD and the application layer as follows:

- The UVD is treated as an independent object (or a library). The application layer is completely unaware of its functions.

- The only interaction between the UVD and the application layer is achieved through the input bit-stream buffer and the output video frame buffer as shown in the following flow graph (Fig. 9).

![Figure 9: Interaction between UVD and Application Environment](image)

The application environment and the UVD operate independently. The application environment is responsible for filling in the bit stream buffer and removing frames from the video frame buffer. The left side of the flow graph (Fig. 9) shows the typical functionality of the application environment relating to video decoding. The application typically reads in the video bit-stream (usually de-multiplexed from a multimedia stream) and fills the bit stream buffer. When enough data is available, video frames are removed from the frame buffer. The timing and display control, i.e. the rate and the conditions at which the buffers are filled and emptied, are not a normative part of the FCVC framework as these are dependant on the application. The UVD is responsible for reading the bit-stream if data is available, processing the bit-stream as explained above (see Figure 8) and storing the output in the video frame buffer. In a typical software implementation, the application and the UVD may execute using two independent processing threads.

B. Implementing a decoding process in the DDS

In order to demonstrate DDS coding, Figure 10 shows a simple inverse quantization function of a video decoder written in C. Simplifications to high level C-code are introduced here so that the DDS functional primitives have a one-to-one correspondence with each line of code shown. The first simplification is that the value 2 is declared as a constant since all the values used in the code are stored as part of the data segment in DDS. The second simplification is that all
match SIS instructions. Note that the complexity of the high level code that can be written depends on the parsing tool used to decompose the instructions into functional primitives. This sample function contains processing instructions for arithmetic operations and a call to a natively supported advanced function called “Int_array_mult_int”. This is one of the advanced signal processing specific functions supported by DDS to scale an integer array by an integer value. The first input parameter points to the start of the array, the second parameter is the value, while the third is the array size. The list of currently supported signal processing specific functions can be found at [14].

```c
//Inverse Quantise
void Q4(int * i_blk_y_base, unsigned char blk_size, int q_step) {
    int total_blk_size;
    int tmp;
    const int two = 2;

    total_blk_size = blk_size * blk_size;
    tmp = total_blk_size / two;
    total_blk_size = total_blk_size + tmp;
    Int_array_mult_int(i_blk_y_base, q_step, total_blk_size);
}
```

**Figure 10: A simple C function for inverse quantisation.**

This high level function is fed to the DDS generator, as shown in Figure 2, in order to generate the DDS. Table 5 shows the data in the function placed in a type-specific indexed list, as discussed in Section III.A. Note that the indexes are allocated in ascending order starting from the output parameter, and going through any input parameters to any data elements declared in the function. For example, if the function does not have a return parameter, the index value zero is allocated to the first input parameter.

**Table 5: The Data Segment for the inverse quantisation function.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Index</th>
<th>Data Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>U_CHAR_V</td>
<td>blk_size</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>INT_V</td>
<td>Q_step</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>INT_V</td>
<td>total_blk_size</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>INT_V</td>
<td>tmp</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>INT_V</td>
<td>two</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>INT_P</td>
<td>i_blk_y_base</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 6: The Program Segment for the inverse quantisation function.**

The SIS instructions for the Program Segment, as discussed in Section III.A, are shown in Table 6. The four instructions in the table directly correspond to the four instructions in the high-level code. The instructions indicated by the values in the OPCODE column are shown in parentheses. The first instruction is a generic multiplication instruction where the types of the left-hand-side and right-hand-side operands, referring to Equation (1), are specified using their Type_ID value and index. The second instruction is an “integer division”. Analysis of common video codecs shows that a significant number of operations are carried out using integer arithmetic. Therefore, the DDS is designed to support integer-specific versions of commonly used arithmetic operations. The integer division operation assumes that all operands are of INT_V type and therefore, type information need not be coded, thus saving configuration bandwidth. Furthermore, computational complexity at the decoder is saved because operand type resolutions are not required. The integer addition instruction follows the same principles as integer division. The fourth instruction is a video coding specific (VCS) operation. A VCS operation is handled similar to a function call. The OPCODE specifies that this instruction is a VCS operation and TYPE_O specifies the VCS operation ID which corresponds to the “Int_array_mult_int” operation. Three parameters are passed in to this function. The indexes of these parameters are specified from OUT field on words. Since the VCS operations are strictly typed, the Type_ID’s of the input/output parameters are implied.

The functional processing primitives, that is, the data and program segments, are encapsulated in the transmission syntax to be coded and transmitted to the receiver. The transmission syntax elements for the Inverse Quantise function are shown in Table 7. Refer to the syntax shown in Figure 6 for the new function. Given that the transmission of reconfiguration information implies an expansion of the video bit stream, entropy coding techniques are employed to reduce the DDS overhead by exploiting statistical redundancies in the syntax elements. Some operand types and/or indexes can either be inferred or take only a subset of values depending on the OPCODE, enabling data representations to be significantly shorter than a byte for these fields, e.g., for integer operations, the input and/or output types can be inferred from the OPCODE. Although it is out of the scope of this paper to describe the entropy coding of DDS in detail, selected entropy coding results are presented in the next section.

Finally, assuming that this is the only function transmitted, a complete and decodable DDS stream is created by inserting the syntax elements DDS_Header, DDS_Command (CREATE_FNS) and No_Of_Fns_Minus_One (0).
C. A real-time software prototype

A real-time software prototype has been developed using the Microsoft DirectShow framework. Video is captured from a camera and encoded using a selected initial configuration. The encoder sends a complete description of the decoding process via the DDS prior to sending coded video. A UVD receives and instantiates the DDS and proceeds to decode and display video. In the current prototype, new or modified DDS may be sent once per frame as a header prior to the coded frame data. If a new or modified DDS function is received, the UVD instantiates this new function and carries on decoding the next video frame using the modified decoder structure.

The prototype UVD uses a scripting model where each DDS element is interpreted and executed using corresponding compiled C++ functions. Each decoding function is instantiated as a software object with its own data and program memory. The program memory contains the sequence of processing instructions specified by the DDS. The UVD acts as a virtual processor, stepping through each decoding function object and branching between objects as required, to decode compressed video. Configuration and re-configuration of the UVD is achieved by sending decoding functions, replacing existing functions, for example, replacing one transform with another, or creating new functions which are linked in to the current decoder structure. In the prototype system, re-configuration may occur once per frame.

A simple block-based decoder was implemented using this framework, with functions as illustrated in Figure 11. This block-based decoder is comprised of basic tools, which are: entropy and run-length decoding; inverse quantization; inverse transformation; and, frame reconstruction. The decoder is described using a total of eight functions, including the mandatory “Main” function, F0, with function identifier Fn_ID=0. Note that the 2D Inverse transform function F6 (and 1D transform) is called three times to handle the luminance and two chrominance components of YUV video.

The encoder is configured to use the DCT transform at the beginning of the encoding process. The DDS for the complete decoder, that is, all eight functions, is signaled to the UVD as a header before sending the coded video data or previously instantiated at the receiver. The F7 function contains processing instructions for the inverse DCT. The user interface provided at the encoder allows the user to switch the transform used at the coder to the Haar transform or back to the DCT. Once the Haar transform is selected, the encoder first transmits a replacement function for F7, containing the processing instructions for the inverse Haar transform, labeled as “F7 1D Haar” in Figure 11, and starts encoding the next frame using the Haar transform. Once the new F7 is received by the UVD, the function is replaced, and continues to decode the incoming coded video stream.

Figure 12 shows screenshots of the prototype UVD. The prototype consists of a video encoder and a UVD in which live video is captured, encoded, decoded and displayed. The top picture shows a screenshot of the user interface for the encoder and the display. The user interface allows the user to change the coding configuration, in this case the transform used for encoding. The bottom picture shows the real-time performance data for the UVD. The current decoder structure is illustrated as a tree, where each individual function is shown. Note that the UVD does not have knowledge of function names, therefore, an extra four bytes per function were used as side information to convey a short name for demonstration purposes. The function ID used for Haar transform in this example is different from that of the functional diagram in Figure 11, since the prototype is designed to store any currently unused (not being called) functions which may be called in future reconfigurations. This is demonstrated using a “dummy” function with Fn_ID=3.

---

### Table 7: The DDS for the inverse quantisation function.

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fn_ID</td>
<td>4</td>
<td>Illustrative value</td>
</tr>
<tr>
<td>Prog_Size</td>
<td>4</td>
<td>No. of Instructions</td>
</tr>
<tr>
<td>U_CHAR_V_Size</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>INT_V_Size</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>INT_P_Size</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>[All other data types]</td>
<td>0</td>
<td>Not present</td>
</tr>
<tr>
<td>Instruction(s)</td>
<td>Values in Table 6</td>
<td></td>
</tr>
<tr>
<td>No_Of_Init_Values</td>
<td>0</td>
<td>for U_CHAR_V</td>
</tr>
<tr>
<td>No_Of_Init_Values</td>
<td>1</td>
<td>for INT_V const value “two”</td>
</tr>
<tr>
<td>Index</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>2</td>
<td>a 32-bit number</td>
</tr>
<tr>
<td>No_Of_Init_Pointers</td>
<td>0</td>
<td>for INT_P</td>
</tr>
</tbody>
</table>

---

![Figure 11: Functional diagram of simple decoder with transform reconfiguration.](image-url)
The current software prototype was tested on a PC with an Intel Pentium IV (3 GHz) processor. The video frame size is 320 x 240 pixels in YUV 4:2:0 format. The real-time frame rate was observed to be between 12 – 19 fps, which corresponds to a frame delay between 53 – 83 ms. The maximum reconfiguration delay observed was less than 1 ms and the typical delay was around 0.1 ms, as illustrated in the lower screenshot of Figure 12. Therefore, at present the reconfiguration delay is approximately 1% of the frame delay.

Table 8 shows DDS overhead for different configuration information. For example, “dct.dds” is the DDS for the single DCT function. The table shows the original size of the DDS for DCT and the compressed size after entropy coding. The configuration named “decoder_dct.dds” is the DDS for transmitting the complete video decoder including the DCT transform. Similar data for the Haar transform is also presented. The proposed reconfiguration scheme was implemented in an H.264 codec [15]. As an example of performance, for the QCIF Carphone sequence at 1.5 bpp a gain of approximately 0.5 dB in PSNR is seen over the baseline codec when transmitting the Haar inverse transform function via decoder_haar.dds and including block-level flags to indicate whether the DCT or Haar transform was used on a particular block. Thus, it is evident that complete decoder functionality can be transmitted using DDS with a modest rate overhead.

These results were obtained by transmitting the Haar inverse transform function to the decoder once per set of 100 frames, so that the overhead of 1195 bytes is distributed over a significant subsequence. In general, source content or system resources might vary significantly enough that a reconfiguration of the codec would be beneficial in an RD sense perhaps at most every frame. Thus, we would expect that the frequency of partial reconfiguration would be once per frame at most and, more likely, on the order of a subsequence or video scene.

Table 8: DDS overhead.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DDS size (bytes)</th>
<th>Compressed DDS size (bytes)</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>dct.dds</td>
<td>734</td>
<td>372</td>
<td>50%</td>
</tr>
<tr>
<td>decoder_dct.dds</td>
<td>2874</td>
<td>1351</td>
<td>53%</td>
</tr>
<tr>
<td>haar.dds</td>
<td>465</td>
<td>218</td>
<td>53%</td>
</tr>
<tr>
<td>decoder_haar.dds</td>
<td>2605</td>
<td>1195</td>
<td>54%</td>
</tr>
</tbody>
</table>

V. CONCLUSIONS

In this work we introduced a framework for dynamically configuring or reconfiguring a video decoder. At the heart of this framework is the Decoder Description Syntax which is capable of completely describing the function and structure of a video decoder in terms of processing and control primitives. The DDS is designed for compact representation and efficient implementation on a wide range of processing platforms, with a single, platform-independent DDS description for each potential coding format. This approach allows for adaptation of the coder and decoder to varying source statistics, channel conditions, hardware platforms, or other resource constraints. We have demonstrated a complete, real-time implementation of a video decoder in DDS running on a prototype Universal Video Decoder.

This important new approach opens up a number of research challenges, including the following.

- Increasing the computational performance of the UVD / DDS on software platforms to approach that of conventional, fixed decoder designs. Possible methods include efficient handling of data structures such as arrays and blocks, just-in-time compilation, etc [16].
- Developing new approaches to automatically generating and optimizing DDS to increase rate-distortion performance through adaptive video coding.
- Efficiently implementing the UVD / DDS on a range of software platforms and on hardware platforms, for example through hardware/software co-design techniques for reconfigurable hardware platforms [17,18,19] and through explicit or implicit representation of parallelism.
in the DDS.

- Developing transport protocols to enable the DDS to be used in a variety of transmission scenarios including multi-cast, broadcast, streaming and error-prone transmission environments.

- Ensuring successful DDS implementation on resource-constrained platforms and prevent execution of "malicious" DDS code through limiting the DDS execution environment in terms of memory size, file accesses and processing resources.

- Integrating the proposed DDS into the MPEG2 RVC framework to provide for low-level transport of FUs and coded FU or FU interconnection reconfiguration data as suggested in [20].

We believe that the potential benefits of this flexible and dynamically re-programmable definition syntax are wide ranging and that DDS can provide a platform for exciting new developments in video coding.

REFERENCES
