International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:11, 2009

A Scheme of Model Verification of the Concurrent Discrete Wavelet Transform (DWT) for Image Compression

Kamrul Hasan Talukder and Koichi Harada

***Abstract*—**The scientific community has invested a great deal of effort in the fields of discrete wavelet transform in the last few decades. Discrete wavelet transform (DWT) associated with the vector quantization has been proved to be a very useful tool for the compression of image. However, the DWT is very computationally intensive process requiring innovative and computationally efficient method to obtain the image compression. The concurrent transformation of the image can be an important solution to this problem. This paper proposes a model of concurrent DWT for image compression. Additionally, the formal verification of the model has also been performed. Here the Symbolic Model Verifier (SMV) has been used as the formal verification tool. The system has been modeled in SMV and some properties have been verified formally.

***Keywords*—**Computation Tree Logic, Discrete Wavelet Transform, Formal Verification, Image Compression, Symbolic Model Verifier.

I. Introduction

THE research in compression techniques has stemmed from the ever increasing need for efficient data transmission, storage and utilization of hardware resources. Uncompressed image data require considerable storage capacity and transmission bandwidth. Despite rapid progresses in mass storage density, processor speeds and digital communication system performance demand for data storage capacity and data transmission bandwidth continues to outstrip the capabilities of available technologies. The recent growth of data intensive multimedia based applications have not only sustained the need for more efficient ways to encode signals and images but have made compression of such signals central to signal storage and digital communication technology.

Compressing an image is significantly different from compressing raw binary data. Of course, general purpose compression programs can be used to compress images, but the result is less than optimal. This is because images have certain statistical properties which can be exploited by encoders specifically designed for them. Also, some of the finer details in the image can be sacrificed for the sake of saving a little more bandwidth or storage space.

Lossless compression involves with compressing data which, when decompressed, will be an exact replica of the original data. This is the case when binary data such as executables documents etc. are compressed. They need to be exactly reproduced when decompressed. On the other hand, images need not be reproduced 'exactly'. An approximation of the original image is enough for most purposes, as long as the error between the original and the compressed image is tolerable.

The neighboring pixels of most of the images are highly correlated and therefore hold redundant information from certain perspective of view [1]. The foremost task then is to find out less correlated representation of the image. Image compression is actually the reduction of the amount of this redundant data (bits) without degrading the quality of the image to an unacceptable level [2] [3] [4]. There are mainly two basic components of image compression - redundancy reduction and irrelevancy reduction. The redundancy reduction aims at removing duplication from the signal source image while the irrelevancy reduction omits parts of the signal that is not noticed by the signal receiver i.e., the Human Visual System (HVS) [5] which presents some tolerance to distortion, depending on the image content and viewing conditions. Consequently, pixels must not always be regenerated exactly as originated and the HVS will not detect the difference between original and reproduced images.

The current standards for compression of still image (e.g., JPEG) use Discrete Cosine Transform (DCT), which represents an image as a superposition of cosine functions with different discrete frequencies [6]. The DCT can be regarded as a discrete time version of the Fourier Cosine series. It is a close relative of Discrete Fourier Transform (DFT), a technique for converting a signal into elementary frequency components. Thus, DCT can be computed with a Fast Fourier Transform (FFT) like algorithm of complexity $O(n\log_2 n)$.

More recently, the wavelet transform has emerged as a cutting edge technology within the field of image analysis. The wavelet transformations have a wide variety of different applications in computer graphics including radiosity [7], multiresolution painting [8], curve design [9], mesh optimization [10], volume visualization [11], image searching [12] and one of the first applications in computer graphics,

Manuscript received November 15, 2007

Kamrul Hasan Talukder is a Graduate student in the Department of Information Engineering of the Graduate School of Engineering in Hiroshima University, Japan. (e-mail: khtalukder@hiroshima-u.ac.jp).

Koichi Harada is a Professor in the Department of Information Engineering of the Graduate School of Engineering in Hiroshima University, Japan. (e-mail: hrd@hiroshima-u.ac.jp).

image compression. The Discrete Wavelet Transformation (DWT) provides adaptive spatial frequency resolution (better spatial resolution at high frequencies and better frequency resolution at low frequencies) that is well matched to the properties of an HVS.

This paper proposes a technique of concurrent DWT based image compression which has also been formally verified. Simulation and testing [13] are some of the traditional approaches for verifying the systems. Simulation and testing both involve making experiments before deploying the system in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. Checking all the possible interactions and finding potential pitfalls using simulation and testing techniques is not always possible. Formal verification [14], an appealing alternative to simulation and testing, conducts an exhaustive exploration of all possible behaviors of the system. Thus, when a design is marked correct by the formal method, it implies that all behaviors have been explored and the question of adequate coverage or a missed behavior becomes irrelevant. There are some robust tools for formal verification such as SMV, SPIN, COSPAN, VIS etc [14]. Our method has been modeled in SMV and the properties of the system have been verified formally.

II. DWT in Image Compression

Wavelet transform exploits both the spatial and frequency correlation of data by dilations (or contractions) and translations of mother wavelet on the input data. It supports the multiresolution analysis of data i.e. it can be applied to different scales according to the details required, which allows progressive transmission and zooming of the image without the need of extra storage. Another encouraging feature of wavelet transform is its symmetric nature that is both the forward and the inverse transform has the same complexity, building fast compression and decompression routines. Its characteristics well suited for image compression include the ability to take into account of Human Visual System's (HVS) characteristics, very good energy compaction capabilities, robustness under transmission, high compression ratio etc.

The implementation of wavelet compression scheme is very similar to that of subband coding scheme: the signal is decomposed using filter banks. The output of the filter banks is down-sampled, quantized, and encoded. The decoder decodes the coded representation, up-samples and recomposes the signal.

Wavelet transform divides the information of an image into approximation and detail subsignals. The approximation subsignal shows the general trend of pixel values and other three detail subsignals show the vertical, horizontal and diagonal details or changes in the images. If these details are very small (threshold) then they can be set to zero without significantly changing the image. The greater the number of zeros the greater the compression ratio. If the energy retained (amount of information retained by an image after compression and decompression) is 100% then the compression is lossless as the image can be reconstructed exactly. This occurs when the threshold value is set to zero, meaning that the details have not been changed. If any value is changed then energy will be lost and thus lossy compression occurs. As more zeros are obtained, more energy is lost. Therefore, a balance between the two needs to be found out [15].

The primary aim of any compression method is generally to express an initial set of data using some smaller set of data either with or without loss of information. As for an example, let we have a function $f(x)$ expressed as a weighted sum of basis function $u_1(x),........,u_m(x)$ as given below-

$$f(x) = \sum_{i=1}^{m} c_i u_i(x)$$

where $c_1,..............,c_m$ are some coefficients. We here will try to find a function that will approximate $f(x)$ with smaller coefficients, perhaps using different basis. That means we are looking for-

$$\hat{f}(x) = \sum_{i=1}^{\hat{m}} \hat{c}_i \hat{u}_i(x)$$

with a user-defined error tolerance ε ($\varepsilon = 0$ for lossless compression) such that $m > \hat{m}$ and $\left\| f(x) - \hat{f}(x) \right\| \le \varepsilon$. In general, one could attempt to construct a set of basis functions $\hat{u}_1,............,\hat{u}_{\hat{m}}$ that would provide a good approximation in a fixed basis.

One form of the compression problem is to order the coefficients $c_1,........,c_m$ so that for $m > \hat{m}$, the first $\hat{m}$ elements of the sequence give the best approximation $\hat{f}(x)$ to $f(x)$ as measured in the L^2 form.

Let $\pi(i)$ be a permutation of $1,.......,m$ and $\hat{f}(x)$ be a function that uses the coefficients corresponding to the first $\hat{m}$ numbers of the permutation $\pi(i)$:

$$\hat{f}(x) = \sum_{i=1}^{\hat{m}} c_{\pi(i)} u_{\pi(i)}$$

The square of the L^2 error in this approximation is given by-

$$\begin{aligned} \left\| f(x) - \hat{f}(x) \right\|_2^2 &= \left\langle f(x) - \hat{f}(x) \mid f(x) - \hat{f}(x) \right\rangle \\ &= \left\langle \sum_{i=\hat{m}+1}^{m} c_{\pi(i)} u_{\pi(i)} \mid \sum_{j=\hat{m}+1}^{m} c_{\pi(i)} u_{\pi(i)} \right\rangle \\ &= \sum_{i=\hat{m}+1}^{m} \sum_{j=\hat{m}+1}^{m} c_{\pi(i)} c_{\pi(i)} \left\langle u_{\pi(i)} \mid u_{\pi(i)} \right\rangle \\ &= \sum_{i=\hat{m}+1}^{m} (c_{\pi(i)})^2 \end{aligned}$$

Wavelet image compression using the L^2 norm can be summarized in the following ways:

i) Compute coefficients $c_1,......,c_m$ representing an image in a normalized two-dimensional Haar basis.

ii) Sort the coefficients in order of decreasing magnitude to produce the sequence $c_{\pi(1)},..........,c_{\pi(m)}$.

iii) Given an allowable error ε and starting from $\hat{m} = m$, find the smallest $\hat{m}$ for which

$$\sum_{i=\hat{m}+1}^{m} \left(c_{\pi(i)}\right)^2 \leq \varepsilon^2$$

The first step is accomplished by applying either of the 2D Haar wavelet transforms being sure to use normalized basis functions. Any standard sorting method will work for the second step and any standard search technique can be used for third step. However, for large images sorting becomes exceedingly slow. The procedure below outlines a more efficient method of accomplishing steps 2 and 3, which uses a binary search strategy to find a threshold τ below which coefficients can be truncated.

The procedure takes as input a 1D array of coefficients *c* (with each coefficient corresponding to a 2D basis function) and an error tolerance ε. For each guess at a threshold τ the algorithm computes the square of the L^2 error that would result from discarding coefficients smaller in magnitude than τ. This squared error *s* is compared to ε^2 at each loop to decide if the search would continue in the upper or lower half of the current interval. The algorithm halts when the current interval is so narrow that the number of coefficients to be discarded no longer changes [16].

```
procedure Compress (C : array [1. .m] of reals; ε : real)
  τmin←min{|c[i]|}
  τmax←max{|c[i]|}
  do
     τ ←(τmin + τmax)/2
     s ← 0
     for i ← 1 to m do
          if |C [i]| < τ then s ← s + |C [i]|^2
     end for
     if s < ε^2 then τmin← τ else τmax← τ
  until τmin≈ τmax
  for i ← 1 to m do
       if |C [i]| < τ then C [i] ← 0
   end for
end procedure
```

The below pseudocode fragment for a greedy L^1 compression scheme, which works by accumulating in a 2D array $\Delta[x,y]$ the error introduced by discarding a coefficient and checking if this error has exceeded a user-defined threshold.

```
for each pixel (x,y) do
     Δ [x,y] ← 0
end for
for i← 1 to m do
     Δ' ← Δ + error from discarding c[i]
     if Σ_{x,y} |Δ'[x, y]| < ε then
          c[i] ← 0
          Δ ← Δ'
     end if
end for
```

To understand how wavelets work, let us start with a simple example. Assume we have a 1D image with a resolution of four pixels, having values [9 7 3 5]. Haar wavelet basis can be used to represent this image by computing a wavelet transform. To do this, first average the pixels together, pairwise, is calculated to get the new lower resolution image with pixel values [8 4]. Clearly, some information is lost in this averaging process. We need to store some *detail coefficients* to recover the original four pixel values from the two averaged values. In our example, 1 is chosen for the first detail coefficient, since the average computed is 1 less than 9 and 1 more than 7. This single number is used to recover the first two pixels of our original four-pixel image. Similarly, the second detail coefficient is -1, since 4 + (-1) = 3 and 4 - (-1) = 5. Thus, the original image is decomposed into a lower resolution (two-pixel) version and a pair of detail coefficients. Repeating this process recursively on the averages gives the full decomposition shown in Table I:

TABLE I: DECOMPOSITION TO LOWER RESOLUTION

Resolution	Averages	Detail Coefficients
4	[9 7 3 5]	
2	[8 4]	[1 -1]
1	[6]	[2]

Thus, for the one-dimensional Haar basis, the wavelet transform of the original four-pixel image is given by [6 2 1 -1]. The way used to compute the wavelet transform by recursively averaging and differencing coefficients, is called a *filter bank*. We can reconstruct the image to any resolution by recursively adding and subtracting the detail coefficients from the lower resolution versions.

It has been shown how one dimensional image can be treated as sequences of coefficients. Alternatively, we can think of images as piecewise constant functions on the half-open interval [0, 1). To do so, the concept of a *vector space* is used. A one-pixel image is just a function that is constant over the entire interval [0, 1). Let V^0 be the vector space of all these functions. A two pixel image has two constant pieces over the intervals [0, 1/2) and [1/2, 1). We call the space containing all these functions V^1. If we continue in this manner, the space V^j will include all piecewise-constant functions defined on the interval [0, 1) with constant pieces over each of 2^j equal subintervals. We can now think of every one-dimensional image with 2^j pixels as an element, or vector, in V^j. Note that because these vectors are all functions defined on the unit interval, every vector in V^j *is* also contained in V^{j+1}. For example, we can always describe a piecewise constant function with two intervals as a piecewise-constant function with four intervals, with each interval in the first function corresponding to a pair of intervals in the second. Thus, the spaces V^j are nested; that is, $V^0 \subset V^1 \subset V^2 \subset \ldots\ldots$ This nested set of spaces V^j is a necessary ingredient for the mathematical theory of multiresolution analysis [16]. It guarantees that

every member of V^0 can be represented exactly as a member of higher resolution space V^1. The converse, however, is not true: not every function $G(x)$ in V^1 can be represented exactly in lower resolution space V^0; in general there is some lost detail [17].

Now we define a basis for each vector space V^j. The basis functions for the spaces V^j are called *scaling functions*, and are usually denoted by the symbol ϕ. A simple basis for V^j is given by the set of scaled and translated box functions [18]:

$$\phi_i^j(x) := \phi(2^j x - i) \quad i = 0, 1, 2.....2^j - 1 \text{ where}$$

$$\phi(x) := \begin{cases} 1 & \text{for } 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

The wavelets corresponding to the box basis are known as the *Haar wavelets*, given by-

$$\Psi_i^j(x) := \Psi(2^j x - i) \quad i = 0, 1, 2.....2^j - 1 \text{ where}$$

$$\Psi(x) := \begin{cases} 1 & \text{for } 0 \le x < 1/2 \\ -1 & \text{for } 1/2 \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

Thus, the DWT for an image as a 2D signal will be obtained from 1D DWT. We get the scaling function and wavelet function for 2D by multiplying two 1D functions. The scaling function is obtained by multiplying two 1D scaling functions: $\phi(x,y)=\phi(x)\phi(y)$. The wavelet functions are obtained by multiplying two wavelet functions or wavelet and scaling function for 1D. For the 2D case, there exist three wavelet functions that scan details in horizontal $\Psi^{(1)}(x,y)= \phi(x)\Psi(y)$, vertical $\Psi^{(2)}(x,y)= \Psi(x)\phi(y)$ and diagonal directions: $\Psi^{(3)}(x,y)= \Psi(x)\ \Psi(y)$. This may be represented as a four channel perfect reconstruction filter bank as shown in Fig. 1. Now, each filter is 2D with the subscript indicating the type of filter (HPF or LPF) for separable horizontal and vertical components. By using these filters in one stage, an image is decomposed into four bands. There exist three types of detail images for each resolution: horizontal (HL), vertical (LH), and diagonal (HH). The operations can be repeated on the low low (LL) band using the second stage of identical filter bank. Thus, a typical 2D DWT, used in image compression, generates the hierarchical structure shown in Fig. 2.

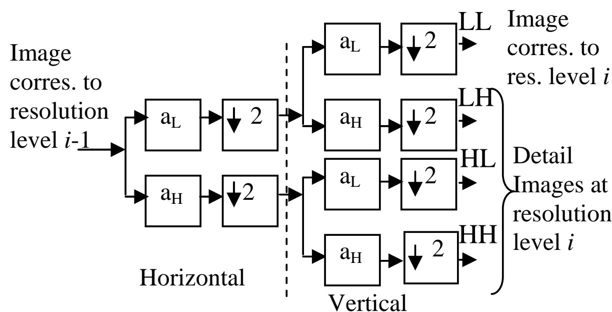

Fig. 1 One Filter Stage in 2D DWT

<table>
<tr><td>LL</td><td>HL3</td><td rowspan="2">HL2</td><td rowspan="3">HL1</td></tr>
<tr><td>LH3</td><td>HH3</td></tr>
<tr><td colspan="2">LH2</td><td>HH2</td></tr>
<tr><td colspan="3">LH1</td><td>HH1</td></tr>
</table>

Fig. 2 Structure of wavelet decomposition

The transformation of the 2D image is a 2D generalization of the 1D wavelet transformed already discussed. It applies the 1D wavelet transform to each row of pixel values. This operation provides us an average value along with detail coefficients for each row. Next, these transformed rows are treated as if they were themselves an image and apply the 1D transform to each column. The resulting values are all detail coefficients except a single overall average co-efficient. In order to complete the transformation, this process is repeated recursively only on the quadrant containing averages.

Now let us see how the 2D Haar wavelet transformation is performed. The image is comprised of pixels represented by numbers [19]. Consider the 8×8 image taken from a specific portion of a typical image shown in Fig. 3. The matrix (a 2D array) representing this image is shown in Fig. 4.

Now we perform the operation of averaging and differencing to arrive at a new matrix representing the same image in a more concise manner. Let us look how the operation is done. Consider the first row of the Fig. 4.

Averaging: (64+2)/2=33, (3+61)/2=32, (60+6)/2=33, (7+57)/2=32

Differencing: 64–33 =31, 3–32= –29, 60–33=27 and 7–32= –25

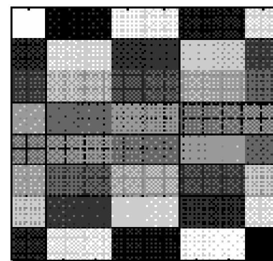

Fig. 3 A 8×8 image

$$\begin{pmatrix} 64 & 2 & 3 & 61 & 60 & 6 & 7 & 57 \\ 9 & 55 & 54 & 12 & 13 & 51 & 50 & 16 \\ 17 & 47 & 46 & 20 & 21 & 43 & 42 & 24 \\ 40 & 26 & 27 & 37 & 36 & 30 & 31 & 33 \\ 32 & 34 & 35 & 29 & 28 & 38 & 39 & 25 \\ 41 & 23 & 22 & 44 & 45 & 19 & 18 & 48 \\ 49 & 15 & 14 & 52 & 53 & 11 & 10 & 56 \\ 8 & 58 & 59 & 5 & 4 & 62 & 63 & 1 \end{pmatrix}$$

Fig. 4 2D array representing the Fig. 3

So, the transformed row becomes (33 32 33 32 31 –29 27 –25). Now the same operation on the average values i.e. (33 32 33 32) is performed. Then we perform the same operation on the averages i.e. first two elements of the new transformed

row. Thus the final transformed row becomes (32.5 0 0.5 0.5 31 –29 27 –25). The new matrix we get after applying this operation on each row of the entire matrix of Fig. 4. is shown in Fig. 5. Performing the same operation on each column of the matrix in Fig. 5, we get the final transformed matrix as shown in Fig. 6. This operation on rows followed by columns of the matrix is performed recursively depending on the level of transformation meaning the more iteration provides more transformations. Note that the left-top element of the Fig. 6 i.e. 32.5 is the only averaging element which is the overall average of all elements of the original matrix and the rest all elements are the details coefficients. The main part of the C program used to transform the matrix is shown in Fig. 7. The 2D array *mat* holds the values which represent the image.

The point of the wavelet transform is that regions of little variation in the original image manifest themselves as small or zero elements in the wavelet transformed version. The 0's in the Fig. 6 are due to the occurrences of identical adjacent elements in the original matrix. A matrix with a high proportion of zero entries is said to be *sparse*. For most of the image matrices, their corresponding wavelet transformed versions are much sparser than the originals. Very sparse matrices are easier to store and transmit than ordinary matrices of the same size. This is because the sparse matrices can be specified in the data file solely in terms of locations and values of their non-zero entries.

$$\begin{pmatrix} 32.5 & 0 & 0.5 & 0.5 & 31 & -29 & 27 & -25 \\ 32.5 & 0 & -0.5 & -0.5 & -23 & 21 & -19 & 17 \\ 32.5 & 0 & -0.5 & -0.5 & -15 & 13 & -11 & 9 \\ 32.5 & 0 & 0.5 & 0.5 & 7 & -5 & 3 & -1 \\ 32.5 & 0 & 0.5 & 0.5 & -1 & 3 & -5 & 7 \\ 32.5 & 0 & -0.5 & -0.5 & 9 & -11 & 13 & -15 \\ 32.5 & 0 & -0.5 & -0.5 & 17 & -19 & 21 & -23 \\ 32.5 & 0 & 0.5 & 0.5 & -25 & 27 & -29 & 31 \end{pmatrix}$$

Fig. 5 Array after operation on each row of Fig. 4

$$\begin{pmatrix} 32.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & -4 & 4 & -4 \\ 0 & 0 & 0 & 0 & 4 & -4 & 4 & -4 \\ 0 & 0 & 0.5 & 0.5 & 27 & -25 & 23 & -21 \\ 0 & 0 & -0.5 & -0.5 & -11 & 9 & -7 & 5 \\ 0 & 0 & 0.5 & 0.5 & -5 & 7 & -9 & 11 \\ 0 & 0 & -0.5 & -0.5 & 21 & -23 & 25 & -27 \end{pmatrix}$$

Fig. 6 Final Transformed Matrix after one step

It can be seen that in the final transformed matrix, we find a lot of entries zero. From this transformed matrix, the original matrix can be easily calculated just by the reverse operation of averaging and differencing i.e. the original image can be reconstructed from the transformed image without the loss of information. Thus, it yields a lossless compression of the image. However, to achieve more degree of compression, we have to think of the lossy compression. In this case, a nonnegative threshold value say ε is set. Then, any detailed coefficient in the transformed data, whose magnitude is less than or equal to ε, is set to zero. It will increase the number of 0's in the transformed matrix and thus the level of compression is increased. So, $\varepsilon = 0$ is used for a lossless compression. If the lossy compression is used, the approximations of the original image can be built up. The setting of the threshold value is very important as there is a tradeoff between the value of ε and the quality of the compressed image. Finding out an appropriate value of ε is an interesting area to research on. Loosely saying, the compression ratio of the image is calculated by- the number of nonzero elements in original matrix: the number of nonzero elements in updated transformed matrix [20].

```
/*row transformation*/
for(i=0;i<row;i++){w=col;
   do{ k=0;
/*averaging*/     for(j=0;j<w/2;j++)
                       a[j]=((mat[i][j+j]+mat[i][j+j+1])/2);
/*differencing*/ for(j=w/2;j<w;j++,k++)
                       a[j]=mat[i][j-w/2+k]-a[k];
           for(j=0;j<row;j++) mat[i][j]=a[j];
           w=w/2;
   }while(w!=1);
}
/*column transformation*/
for(i=0;i<col;i++){ w=row;
   do{k=0;
/*averaging*/     for(j=0;j<w/2;j++)
                     a[j]=((mat[j+j][i]+mat[j+j+1][i])/2);
/*differencing*/for(j=w/2;j<w;j++,k++)
                     a[j]=mat[j-w/2+k][i]-a[k];
          for(j=0;j<w;j++) mat[j][i]=a[j];
          w=w/2;
   }while(w!=1);
}
```

Fig. 7 The Code

In summary, the main steps of the 2D image compression using wavelet as the basis functions are: (a) Start with the matrix P representing the original image, (b) Compute the transformed matrix T by the operation averaging and differencing (First for each row, then for each column) (c) Choose threshold value ε ($\varepsilon = 0$ for lossless and $\varepsilon =$ some +ve value for lossy) (d) Replace all co-efficient of T which is smaller than or equal to ε by zero. Suppose this matrix is D. (e) Use D to compute the compression ratio and to reconstruct the original image as well.

Now we see the effect of one step averaging and differencing of an image. The Fig. 8 is the original image and the Fig. 9 is the transformed image after applying the one step averaging and differencing. The more steps produce more decomposition.

Fig. 8 Original Image

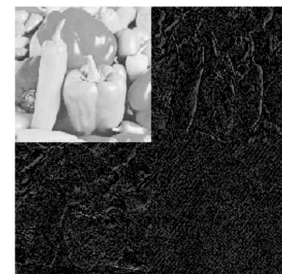

Fig. 9 Transformed Image

III. Concurrent Program Testing

A concurrent program consists of a collection of sequential processes whose execution is interleaved; the interleaving is the result of choices made by a scheduler. Lots of execution interleavings are possible, making testing of all but trivial concurrent programs infeasible.

To make matters worse, functional specifications for concurrent programs often concern intermediate steps of the computation. For example, consider a word-processing program with two processes: one that formats pages and passes them through a queue to the second process, which controls a printer. The functional specification might stipulate that the page-formatter process never deposit a page image into a queue slot that is full and that the printer-control process never retrieve the contents of an empty or partially filled queue slot.

If contemplating the individual execution interleavings of a concurrent program is infeasible, then we must seek methods that allow all executions to be analyzed together. We do have on hand a succinct description of the entire set of executions: the program text itself. Thus, analysis methods that work directly on the program text (rather than on the executions it encodes) have the potential to circumvent problems that limit the effectiveness of testing. For example, here is a rule for showing that some *bad thing* doesn't happen during execution:

Identify a relation between the program variable that is true initially and is left true by each action of the program. Show that this relation implies the "bad thing" is impossible.

Thus, to show that the printer-control process in the previous example never reads the contents of a partially filled queue slot (a *bad thing*), we might see that the shared queue is implemented in terms of two variables:

NextFull points to the queue slot that has been full the longest and is the one the printer-control process will next read.

FirstEmpty points to the queue slot that has been empty the longest and is the one where the page-formatter process will next deposit a page image.

We would then establish that *NextFull* ≠ *FirstEmpty* is true initially and that no action of either process falsifies it. And, from the variable definitions, we would note that *NextFull* ≠ *FirstEmpty* implies that the printer-control process reads the contents of a different queue slot than the page-formatter process writes, so the "bad thing" cannot occur.

It turns out that all functional specifications for concurrent programs can be partitioned into *bad things* and *good things*. Thus, a rule for such *good things* will complete the picture. To show that some *good thing* does happen during execution: *Identify an expression involving the program variables that when equal to some minimal value implies that the "good thing" has happened. Show that this expression (a) is decreased by some program actions that must eventually run, and (b) is not increased by any other program action.*

Note our rules for *bad things* and *good things* do not require checking individual process interleavings. They require only effort proportional to the size of the program being analyzed. Even the size of a large program need not be an impediment—large concurrent programs are often just small algorithms in disguise. Such small concurrent algorithms can be programmed and analyzed; we build a model and analyze it to gain insight about the full-scale artifact [21].

Thus, the correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems.That is why, writing correct concurrent programs is harder than writing sequential ones. This is because the set of potential risks and failure modes is larger - anything that can go wrong in a sequential program can also go wrong in a concurrent one, and with concurrency comes additional hazards not present in sequential programs such as race conditions, data races, deadlocks, missed signals etc.

In some concurrent computing systems communication between the concurrent components is hidden from the programmer, while in others it must be handled explicitly. Explicit communication can be divided into two classes:

Shared Memory Communication: Concurrent components communicate by altering the contents of shared memory locations (exemplified by Java). This style of concurrent programming usually requires the application of some form of locking (e.g. mutual exclusion) to coordinate between multiple threads.

Message Passing Communication: Concurrent components communicate by exchanging messages (exemplified by Erlang). The exchange of messages may be carried out asynchronously (sometimes referred to as "send and pray", although it is standard practice to resend messages that are not acknowledged as received), or may use a style in which the sender blocks until the message is received. Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust, although slower, form of concurrent programming.

Testing concurrent programs is also harder than testing sequential ones. This is trivially true: tests for concurrent programs are themselves concurrent programs. But it is also true for another reason: the failure modes of concurrent programs are less predictable and repeatable than for sequential programs. Failures in sequential programs are deterministic; if a sequential program fails with a given set of inputs and initial state, it will fail every time. Failures in concurrent programs, on the other hand, tend to be rare probabilistic events.

Because of this, reproducing failures in concurrent programs can be difficult. Not only might the failure be rare, and therefore not manifest itself frequently, but it might not occur at all in certain platform configurations, so that bug that happens daily at customer's site might never happen at all in test lab. Further, attempts to debug or monitor the program can introduce timing or synchronization artifacts that prevent the bug from appearing at all. As in Heisenberg's uncertainty principle, observing the state of the system may in fact change it.

IV. The Concurrent Compression System

This section describes how the image is concurrently transformed, the problem lies in the concurrent transformation, the model of the system, the model verification and the result obtained.

A. Concurrent Transformation of Image

We know that wavelet transformation entails transformation of image data horizontally first and then vertically. Here we divide the image plane into *n* horizontal sections which are horizontally transformed *concurrently*. After then the image is divided into *n* vertical sections which are then vertically transformed *concurrently*. It is not a must that the number of horizontal sections is equal to the number of vertical sections. Fig. 10 below illustrates the method.

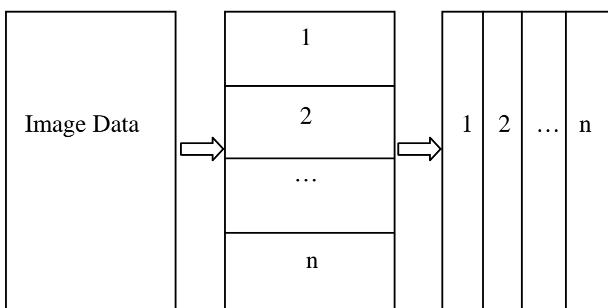

Fig. 10 Image Division

But the problem lies in the concurrency. The system just proposed lets the possibility for vertical transformation to begin on some vertical sections before horizontal transformation in all sections is completed. Vertical sections that are already horizontally transformed can be vertically transformed as illustrated in Fig. 11. That allows the possibility for threads that completed horizontal transformation to go on to vertical transformation without having to wait on other threads to complete horizontal transformation. The gray color indicates sections of image data that are horizontally transformed. The white color indicates sections of image data that are not yet horizontally transformed. The gray vertical section with line stripes can be assigned to a thread for vertical transformation. Before a vertical section is available for transformation, one condition that must be met is that all horizontal sections transform *n* size data horizontally such that an *n* wide vertical section is available with all data points already horizontally transformed.

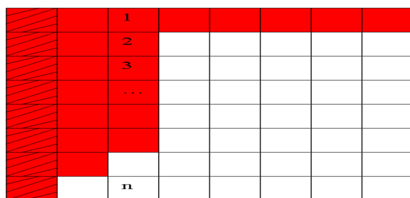

Fig. 11 Ordering the Transformation

The assertion for the verification is that at any time, the vertical transformation does not start on a vertical section that is not horizontally transformed. In Fig. 12, the vertical transformation can start in vertical sections V_0 and V_1 but not in V_2 through V_7.

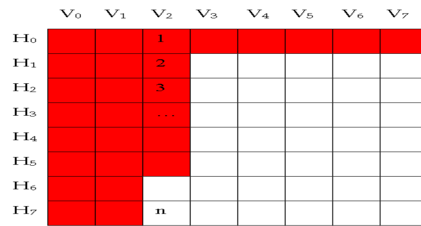

Fig. 12 Vertical vs Horizontal transformation

B. Model Verification

To understand the term “model”, we need to be familiar with transition system and Kripke Structure. A transition system is a structure TS = (S, S0, R) where, S is a finite set of states; S0 $\subseteq$ S is the set of initial states and R $\subseteq$ S × S is a transition relation which must be total i.e. for every s in S there exists s’ in S such that (s, s’) is in R ($\forall$ s $\in$ S $\exists$ s’ $\in$ S . (s, s’) $\in$ R). On the other hand, M = (S, S0, R, AP, L) is a Kripke Structure; where (S, S0, R) is a transition system. AP is a finite set of atomic propositions (each proposition corresponds to a variable in the model) and L is a labeling function. It labels each state with a set of atomic propositions that are true in that state. The atomic propositions and L together convert a transitions system into a model.

The foremost step to verify a system is to specify the properties that the system should have. For example, we may want to show that some concurrent program never deadlocks. These properties are represented by *temporal logic*. Computation Tree Logic (CTL) is one of the versions of *temporal logic*. It is currently one of the popular frameworks used in verifying properties of concurrent systems [22]. Once we know which properties are important, the second step is to construct a *formal model* for that system. The model should capture those properties that must be considered for the establishment of correctness. Model checking includes the traversing the state transition graph (*Kripke Structure*) and of verifying that if it satisfies the formula representing the property or not, more concisely, the system is a model of the property or not.

Each CTL formula is either true or false in a given state of the *Kripke Structure*. Its truth is evaluated from the truth of its sub-formulae in a recursive fashion, until one reaches atomic propositions that are either true or false in a given state. A formula is satisfied by a system if it is true for all the initial states of the system. Mathematically, say, a *Kripke Structure* $K = (S, S0, R, AP, L)$ (system model) and a CTL formula Ψ (specification of the property) are given. We have to determine if $K \models \Psi$ holds (K is a model of Ψ) or not. $K \models \Psi$ holds iff $K, s0 \models \Psi$ for every $s0 \in S0$. If the property does not hold, the model checker will produce a counter example that is an execution path that can not satisfy that formula.

Atomic propositions, standard boolean connectives of propositional logic (e.g., AND, OR, NOT), and temporal

operators all together are used to build the CTL formulae. Each temporal operator is composed of two parts: a path quantifier (universal (***A***) or existential (***E***)) followed by a temporal modality (***F, G, X, U***) and are interpreted relative to an implicit "current state". There are generally many execution paths (the sequences) of state transitions of the system starting at the current state. The path quantifier indicates whether the modality defines a property that should be true of all those possible paths (denoted by universal path quantifier ***A***) or whether the property needs only hold on some path (denoted by existential path quantifier ***E***). The temporal modalities describe the ordering of events in time along an execution path and have the following meaning.

- ***F*** ∅ (reads '∅' holds sometime in the future") is true in a path if there exists a state in that path where formula '∅' is true.
- ***G*** ∅ (reads '∅' holds globally") is true in a path if '∅' is true at each and every state in that path.
- ***X*** ∅ (reads '∅' holds in the next state") is true in a path if '∅' is true in the state reached immediately after the current state in that path.
- ∅ ***U*** φ (reads '∅' holds until 'φ' holds) is true in a path if 'φ' is true in some state in that path, and '∅' holds in all preceding states.

The semantics of the CTL operators are stated below:

- K, s |= EX (Ψ) there exists s' such that s → s' (R(s, s')) and K, s' |=Ψ. It means that s has a successor state s' at which Ψ holds.
- K, s |= EU (Ψ1, Ψ2) iff there exists a path L = s0, s1, … from s and k >= 0 such that: K, L(k) |= Ψ2 and if 0 ≤ j < k, then K, L(j) |= Ψ1.
- K, s |= AU(Ψ1, Ψ2) iff for every path L = s0, s1, … from s there exists k >= 0 such that: K, L(k) |= Ψ2 and if 0 ≤ j < k, then K, L(j) |= Ψ1.
- AX (Ψ): It is not the case there exists a next state at which Ψ does not hold i.e. for every next state Ψ holds.
- EF (Ψ): There exists a path L from s and k >= 0 such that: K, L(k)|=Ψ.
- AG (Ψ): It is not the case there exists a path L from s and k>= 0 such that: K, L(k)|= Ψ i.e. for every path L from s and every k >= 0;K, L(k)|=Ψ
- AF(Ψ) : For every path L from s, there exists k>= 0 such that: K, L(k)|= Ψ.
- EG(Ψ): It is not the case that for every path L from s there is a k >= 0 such that K,L(k)|=Ψ. It means that there exists a path L from s such that, for every k>= 0: K, L(k) |= Ψ.

Some basic CTL operators among those stated above are shown graphically in Fig. 13. In this figure, if it is assumed that in the filled states, the formula *f* holds, then we can say that EF *f*, AF *f*, EG *f*, and AG *f* are satisfied in initial state.

CTL formulas are sometime problematical to interpret. For this, a designer may fail to understand what property has been actually verified. Here we want to add some common constructs of CTL formula used in hardware verification.

- ***AG*** *(Request →* ***AF*** *Acknowledgement)*: For all reachable states (***AG***), if *Request* is asserted in the state, then always at some later point (***AF***), we must reach a state where *Acknowledgement* is asserted. ***AG*** is interpreted relative to the initial states of the system whereas ***AF*** is interpreted relative to the state where *Request* is asserted. A common mistake would be to write *Request →* ***AF*** *Acknowledgement* in place of ***AG*** *(Request →* ***AF*** *Acknowledgement)*. The meaning of the former is that if *Request* is asserted in the initial state, then it is always the case that eventually we reach a state where *Acknowledgement* is asserted, while the latter requires that the condition is true for any reachable state where *Request* holds. If *Request* is identically true, ***AG*** *(Request →* ***AF*** *Acknowledgement)* reduces to *AG AF Acknowledgement*.
- ***AG*** *(****AF*** *DeviceEnabled)*: The proposition *DeviceEnabled* holds infinitely often on every computational path.
- ***AG*** *(****EF*** *start)*: From any reachable state, there must exist a path starting at that state that reaches a state where *start* is asserted. In other words, it must always be possible to reach the restart state.
- ***EF*** *(x ∧* ***EX*** *(x∧* ***EX*** *x)) →****EF*** *(y ∧* ***EX EX*** *z)*: If it is possible for *x* to be asserted in three consecutive states, then it is also possible to reach a state where *y* is asserted and from there to reach in two more steps a state where *z* is asserted.
- ***EF*** *(~Ready ∧ Started)*: It is possible to get to a state where *holds started*, but *ready* does not hold.
- ***AG*** *(Send →* ***A*** *(Send* ***U*** *Receive))*: It is always the case that if *Send* occurs, then eventually Receive is true, and until that time, *Send* must continue to be true.
- ***AG*** *(in →****AX AX AX*** *out)*: Whenever *in* goes high, *out* will go high within three clock cycles.

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:11, 2009

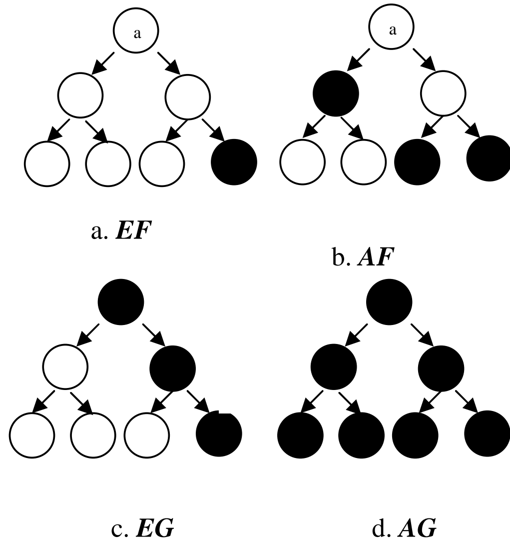

Fig. 13 Basic CTL Operators

C. The Model

Fig. 14 shows the state diagram of the model for verification. It illustrates the tasks of a thread performing horizontal transformation on a horizontal section and vertical transformation on zero or more vertical sections.

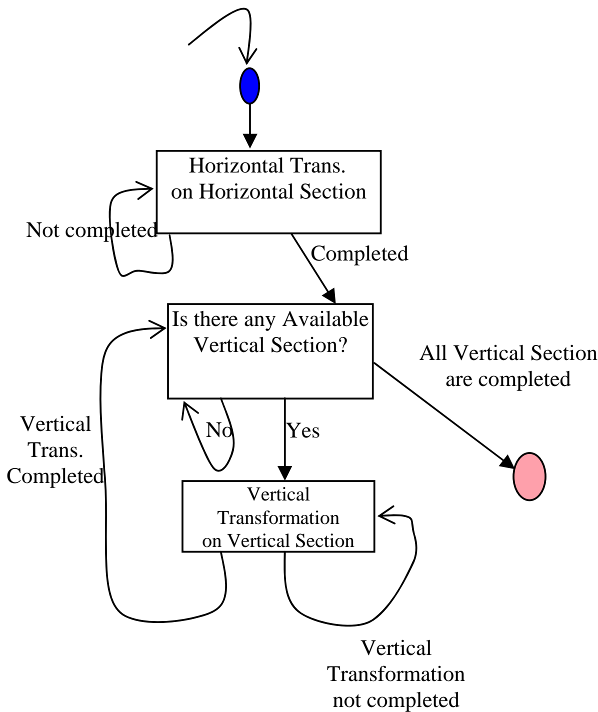

Fig. 14 State diagram of a thread

In order to establish the communication between multiple threads, we require some work to set up and maintain the communication channels. There are several ways to communicate between threads, with some being more efficient than others.

One of the simplest ways to communicate state information between threads is to use a shared object or shared block of memory. A shared object requires very little setup—all we have to do is make sure each thread has a pointer to the object. The object contains whatever custom information we need to communicate between threads, so it should be very efficient.

The second option is the port-based communication. Ports offer a fast and reliable way to communicate between threads and processes on the same or different computers. Ports are also a fairly standard form of communication on many different platforms and their use is well established. In Mac OS X, a port implementation is provided by the Mach kernel. These Mach ports can be used to pass data between processes on the same computer.

The third way is the use of the message queues. The message queues offer an easy-to-use abstraction for thread communication. A message queue is a first-in, first-out (FIFO) queue that manages incoming and outgoing data for the thread. A thread can have both an input and an output queue. The input queue contains work the thread needs to perform, while the output queue contains the results of that work.

To establish communication between the threads, a reliable communication channel is required. Here, this communication channel is modeled as a queue of message, which is the the integral part of the threads. The following figure 15 shows the modeling of the channel as a queue of message from thread 1 to thread 2. The message is pushed through the *tail* of the queue from the thread 1 side and the message is received from the *head* of the queue at the thread 2 side. The Fig. 16 shows the modeling of the communication channel as a queue for the message from thread 2 to thread 1. The message is sent from the thread 2 side and it is received at the *head* of the queue at the thread1 side.

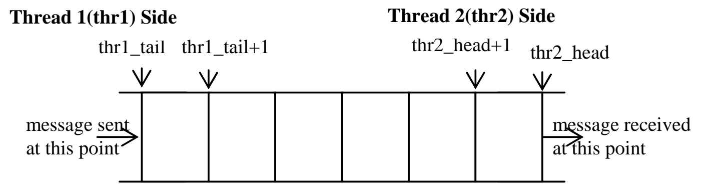

Fig. 15 Channel for message from thread 1to thread 2

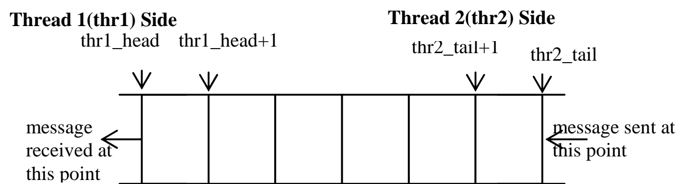

Fig. 16 Channel for reply from thread 2 to thread 1

D. Verification of the Model

The specification for the proposed model verified by the SMV [23] is the **SPEC AG** (**AU** (*hor_trans_count* = *Maxhor*, *bool_vert_trans*) and its result is true. It means that in all states of the transition system it is true that no vertical transformation gets started in any state of all the paths until the variable *hor_trans_count* equals the maximum number of horizontal section i.e. *Maxhor*. This specification is the most

important one that we must get true for the correct wavelet transformation required for the compression of the image.

V. Conclusion

In this paper, we've presented how the discrete wavelet transform is used to image compression, a model for the concurrent wavelet transformation for the compression of the large image, and more importantly the formal verification of the proposed model using the model checking tool SMV that automatically creates a formal environment to efficiently solve the design checking tasks. Some properties of the model have been verified. One of the important properties, in the context of the concurrent DWT transformation, is that at any time, the vertical transformation does not start on a vertical section that is not horizontally transformed which holds true in the model. Perhaps, this is the first time when the concurrent wavelet transformation for the image compression has been formally verified. One of the drawbacks of our modeling in SMV is the smaller size of the queue shared by different threads. In this respect, we hope to use other verification tool like SPIN in future.

References

[1] Jayant, N. and Noll, P., "Digital Coding of Waveforms: Principles and Applications to Speech and Video", NJ: Prentice-Hall, 1984.
[2] Polikar, R. B. "Wavelet Tutorial Part I, II, III, IV", http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html, 2003.
[3] David Salomon, "Data Compression: the Complete Reference", 2nd Ed. Springer, 2002.
[4] Michael B.Martin, "Application of Wavelets to image Compression", M.S. Thesis, Blacksburg Virigina, 1999.
[5] Jayant, N., Johnston, J., and Safranek, R., "Signal compression based on models of human perception", in Proc. IEEE, Vol. 81, October 1993, pp. 1385–1422.
[6] Rao, K. R. and Yip, P., "Discrete Cosine Transform: Algorithms, Advantages and Applications", San Diego, CA: Academic, 1990.
[7] Gortler. S., Schröder, P., Cohen, M., and Hanrahan, P., "Wavelet Radiosity", in Proc. SIGGRAPH, 1993, pp. 221-230.
[8] Berman, D., Bartell, J. and Salesin, D., "Multiresolution Painting and Compositing", in Proc. SIGGRAPH, 1994, pp. 85-90.
[9] Finkelstein. A. and Salesin, D., "Multiresolution Curves", in Proc. SIGGRAPH, 1994, pp. 261-268.
[10] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsberry, M. and Stuetzle, W., "Multiresolution Analysis of Arbitrary Meshes", in Proc. SIGGRAPH, 1995, pp. 173-182.
[11] Lippert, L. and Gross, M., "Fast Wavelet Based Volume Rendering by Accumulation of Transparent Texture Maps", in Proc. EUROGRAPHICS, 1995, pp. 431-443.
[12] Jacobs, C., Finkelstein, A. and Salesin, D., "Fast Multiresolution Image Querying", in Proc. SIGGRAPH, 1995, pp. 277-286.
[13] Myers, Glenford J. "The Art of Software Testing", John Wiley and Sons. ISBN 0-471-04328-1, 1979.
[14] Edmund M. Clakre, Jr. Oma FrumBerg and Doron A. Paled, "Model Checking", The MIT Press, Second Printing, 2000.
[15] Kamrul Hasan Talukder and Koichi Harada, "Development and Performance Analysis of an Adaptive and Scalable Image Compression Scheme with Wavelets", Published in the Proc. of ICICT, BUET, Dhaka, Bangladesh, ISBN: 984-32-3394-8, March 2007, pp. 250-253.
[16] Eric J. Stollnitz, Tony D. Derose and David H. Salesin, "Wavelets for Computer Graphics", Morgan Kaufmann Publishers, Inc., San Francisco. 1996.
[17] Robert L. Cook and Tony DeRose, "Wavelet Noise", ACM Transactions on Graphics, Volume 24, Number 3, Proc. of ACM SIGGRAPH 2005, July 2005, pp. 803-811.
[18] Vetterli, M. and Kovacevic, J., "Wavelets and Subband Coding", Englewood Cliffs, NJ, Prentice Hall, 1995, http://cm.bell-labs.com/who/jelena/Book/home.html
[19] G. Beylkin, R. Coifman, and V. Rokhlin, "Fast wavelet transforms and numerical algorithms", I. Communications on Pure and Applied Mathematics, 44(2), March 1991, pp. 141-183.
[20] Colm Mulcahy, "Image compression using the Haar Wavelet transforms", Spelman Science and Math Journal, Vol 1, No 1, April 1997, pp. 22-31.
[21] Fred B. Schneider, "On Concurrent Programming" Inside Risks 94, CACM 41, 4, Apr 1998.
[22] Michael Huth, "Logic in Computer Science: tool based modeling and reasoning about systems", Proceedings of the International Conference on Frontiers in Education 2000, Kansas City, October 2000.
[23] Cadence Berkeley Laboratories, Free download from the website: http://www.kenmcmil.com/smv.html, Califonia, USA. SMV Model Checker, 1999.