Use of Concurrent Wavelet Transform (WT) to Image Compression and a Verification Analysis of Communicating Threads for Concurrent WT

Kamrul Hasan Talukder and Koichi Harada

Abstract—**Wavelet Transform (WT) has been proved to be a very useful tool for image processing in the recent years. However, WT is very computationally rigorous process requiring novel and computationally competent method to achieve image compression. The concurrent transformation of the image might be one of the solutions to this problem. In this paper, we investigate and analyze the concurrency in communicating threads that perform the wavelet transformation concurrently. The grammar and its syntax are formulated to specify the communication among threads. Moreover, the system modeled in Symbolic Model Verifier (SMV) is formally verified.**

***Index Terms*—** Computation Tree Logic, Discrete Wavelet Transform, Formal Verification, Image Compression, Message Sequence Charts, Symbolic Model Verifier.

I. Introduction

The display, storage and transportation of digital images have moved from obscurity to the commonplace in the last few decades. Now-a-days, many people interact with digital imagery, in one form or another, on a daily basis. The amount of data to be transported demands the development and improvement of the compression approaches. A certain amount of compression is possible without loss of information using *lossless* methods. However, many applications call for an order of magnitude more reduction of size than is possible to meet using such methods. Once the realm of *lossy* compression is entered, a trade-off in size vs. distortion must be made. The fundamental problem is to meet the competing goals of accurate approximation and reduction in storage requirements.

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.

Manuscript received January 7, 2008.

Kamrul Hasan Talukder is a PhD student of the Dept. of Information Engg. in Hiroshima University, Japan. (E-mail: khtalukder@hiroshima-u.ac.jp).

Koichi Harada is a professor of the Dept. of Information Engg. in Hiroshima University, Japan. (e-mail: hrd@hiroshima-u.ac.jp).

The initial breakthrough in the compression of one-dimensional signals [1] was easily extended to the image domain by concatenating image rows or columns into a single stream. Some techniques such as Shannon Fano coding [2] and Huffman coding [3] [4] [5] [6] [7] [8] use redundancy-reduction mechanisms which result in shorter codes for more frequently appearing samples. It is necessary to scan the data samples in order to calculate their probabilities of occurrence and create an exact code. Adaptive variations of these techniques initially assume equal probability for all samples and calculate subsequent probability measures based on a fixed window length prior to the sample of interest. This allows local changes in probability measurements and achieves higher global compression. Run-length coding [9] is another redundancy-reduction coding method where in a scan-line each run of symbols is coded as a pair that specifies the symbol and the length of the run. While most redundancy-reduction methods are lossless, other arbitrarily lossy coding methods have achieved higher levels of compression. Transform coding [10], subband coding [11] [12] [13] [14], vector quantization [15] [16] [17] and predictive coding [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] are among the ones that have achieved high levels of lossy compression. The major transform coding techniques include cosine/sine [28], Fourier [29], Hadamard [30], Haar [30] [10], slant [10] and principal-component (Karhunen-Loeve) transforms [31]. All transform coding based compression algorithms are pixel-based approaches which decompose a signal unto an orthonormal basis to achieve energy compaction. Lossy compression is then achieved by coding the high energy components and leaving out the low energy ones. While some transformations such as the Hadamard and Haar transforms can be performed relatively quickly, other transforms are computationally intensive and either require dedicated hardware or restrictions such as limiting the size of the transform to a power of two. Recently, second-generation compression algorithms based on human visual behavior [32] have been proposed which have the potential for much higher compression ratios.

In the recent years, 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 [33], multiresolution painting [34], curve design [35], mesh optimization [36], volume visualization [37], image searching [38] and one of the first applications in computer graphics, 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.

However, the DWT is very computationally intensive process which requires innovative and computationally efficient method to obtain the image compression. The concurrent transformation might be a useful solution to this problem. In this paper, we investigate the concurrency in wavelet transformation for the compression of image. A verification of the system is also done.

Simulation and testing [39] 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 [40], 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 [40]. The method has been modeled in SMV and the properties of the system have been verified formally.

II. Compression Technique

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-

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

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

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

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 [41].

```
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
```

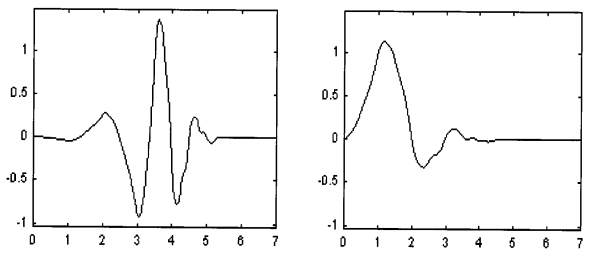
```
    if s < ε² then τmin ← τ else τmax ← τ
  until τmin ≈ τmax
  for i ← 1 to m do
      if |C [i]| < τ then C [i] ← 0
   end for
end procedure
```

The below pseudocode is the 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 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
```

III. Wavelet Transformation for Image Compression

Wavelet transform (WT) represents an image as a sum of wavelet functions (wavelets) with different locations and scales [41]. Any decomposition of an image into wavelets involves a pair of waveforms: one to characterize the high frequencies corresponding to the detailed parts of an image (wavelet function) and one for the low frequencies or smooth parts of an image (scaling function). Fig. 1 shows two waveforms of a family discovered in 1980 by Daubechies: the left one can be used to represent smooth parts of the image and the right one to represent detailed parts of the image. The two waveforms are translated and scaled on the time axis to generate a set of wavelet functions at different locations and on diverse scales. Each wavelet possesses the same number of cycles, such that, the wavelet gets longer while frequency reduces. Low frequencies are transformed with long functions (high scale). High frequencies are transformed with short functions (low scale). The analyzing wavelet is shifted over the full domain of the analyzed function. The result of WT is a set of wavelet coefficients, which measure the contribution of the wavelets at these locations and scales.



Scaling function *phi* Wavelet function *psi*

Fig. 1. Scaling and wavelet function.

WT performs multiresolution image analysis [42]. The result of multiresolution analysis is simultaneous image representation on different resolution levels [43]. The resolution is determined by the specified threshold below which all details are overlooked. The difference between two neighboring resolutions represents details. Therefore, an image can be represented by a low-resolution image (approximation or average part) and the details on each higher resolution level. Let us consider a one dimensional function $f(t)$. The approximation of the function $f(t)$ at the resolution level j is defined as $f_j(t)$. At the one level upper resolution $j+1$, the approximation of the function $f(t)$ is represented by $f_{j+1}(t)$. The details denoted by $d_j(t)$ are included in $f_{j+1}(t)$: $f_{j+1}(t) = f_j(t) + d_j(t)$. This procedure is repetitive for several times and the function can be written as-

$$f(t) = f_j(t) + \sum_{k=j}^{n} d_k(t)$$

In the same way, the space of square integrable functions $L^2(R)$ can be treated as a composition of scaling subspaces V_j and wavelet subspaces W_j such that the approximation of $f(t)$ at resolution $j(f_j(t))$ is contained in V_j and the details $d_j(t)$ are in W_j. V_j and W_j are defined in terms of dilates and translates of scaling function Φ and wavelet function Ψ: $V_j = \{\Phi(2^j x - k) | k \in Z\}$ and $W_j = \{\Psi(2^j x - k) | k \in Z\}$. V_j and W_j are localized in scaled frequency *octaves* by the scale or resolution parameter 2^j and localized spatially by translation k. The scaling subspace V_j must be contained in all subspaces on higher resolutions ($V_j \subset V_{j+1}$). The wavelet subspaces W_j fill the gaps between successive scales: $V_{j+1} = V_j \oplus W_j$. We can start with an approximation on some scale V_0 and then use wavelets to fill in the missing details on finer scales. The finest resolution level includes all square integrable functions-

$$L^2(R) = V_0 + \bigoplus_{j=0}^{\infty} W_j$$

Since $\Phi \in V_0 \subset V_1$, it follows that the scaling function for multiresolution approximation can be found out as the solution to a two-scale dilational equation-

$$\Phi(x) = \sum_k a_L(k) \Phi(2x - k)$$

for some suitable sequence of coefficients $a_L(k)$. Once it has been found, an associated mother wavelet is given by a similar looking formula-

$$\Psi(x) = \sum_k a_H(k) \Phi(2x - k)$$

One of the big discoveries for wavelet analysis was that perfect reconstruction filter banks could be formed using the coefficient sequences $a_L(k)$ and $a_H(k)$ as shown in Fig. 2. The input sequence x is convolved with high-pass (HPF) and low-pass (LPF) filters $a_H(k)$ and $a_L(k)$ and each result is

downsampled by two, yielding the transform signals x_H and x_L. The signal is reconstructed through upsampling and convolution with high and low synthesis filters $s_H(k)$ and $s_L(k)$. For properly designed filters, the signal x is reconstructed exactly ($y = x$).

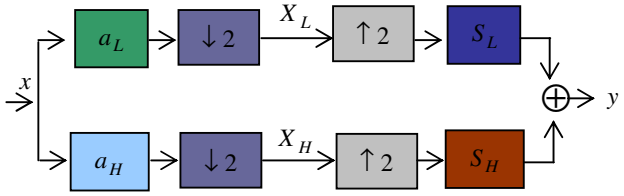


Fig. 2. Two-channel Filter Bank

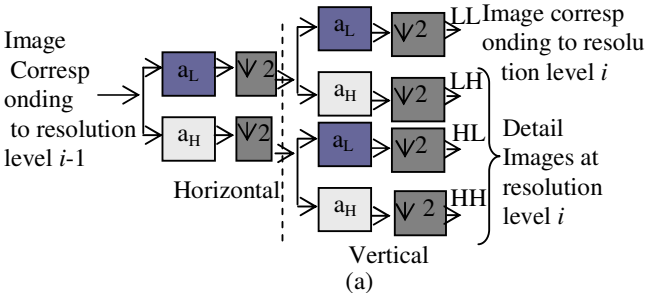
The choice of filter not only decides if the perfect reconstruction is possible or not, it also determines the shape of wavelet to be used to perform the analysis. By cascading the analysis filter bank with itself several times, a digital signal decomposition with dyadic frequency scaling known as DWT can be formed. An efficient way to implement this scheme using filters was developed by Mallat [43]. The new twist that wavelets bring to filter banks is connection between multiresolution analysis and digital signal processing performed on discrete, sampled signals.

The DWT for an image as a 2D signal can be obtained from 1D DWT. The easiest way for finding scaling and wavelet function for 2D is by multiplying two 1D functions. The scaling function for 2D DWT can be obtained by multiplying two 1D scaling functions: $\Phi(x, y) = \Phi(x).\Phi(y)$. Wavelet functions for 2D DWT can be found by multiplying two wavelet functions or wavelet and scaling function for 1D analysis. 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. 3. Now, each filter is 2D with the subscript indicating the type of filter (HPF or LPF) for separable horizontal and vertical components. The resulting four transform components consist of all possible combinations of high and low pass filtering in the two directions. By using these filters in one stage, an image can be decomposed into four bands. There are 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 as shown in Fig. 3(b).

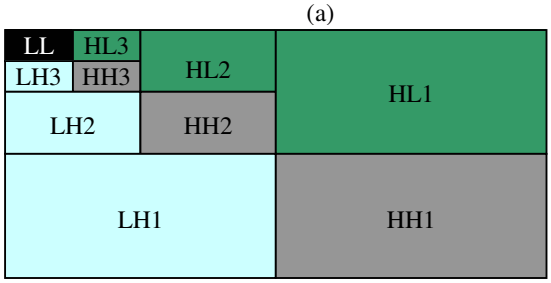
Wavelet multiresolution and direction selective decomposition of images is matched to an HVS [44]. In the spatial domain, the image can be considered as a composition of information on a number of different scales. A wavelet transforms measures gray-level image variations at different scales. In the frequency domain, the contrast sensitivity function of the HVS depends on frequency and orientation of the details.

IV. Concurrency in Wavelet Transformation for Compression 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



(a)



(b)

Fig. 3. One Filter Stage in 2D DWT

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.

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. 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. 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.

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.

A. Communication among threads

A message passing library providing two levels of abstraction namely channel and topology has been developed, for the communication that occurs among threads. These message passing classes can be the part of a larger system that provides a class library for threads, thread synchronization, and message passing. We have used the Message Sequence Charts (MSCs) to visualize the interactions among the threads.

***Message Sequence Charts* (*MSCs*):**

Message Sequence Charts (MSCs) are an attractive visual formalism that is often used in the early stage of system design to specify the system requirements. A main advantage

of an MSC is its clear graphical layout which immediately gives an intuitive understanding of the described system behavior [45]. MSCs are particularly suited to describe the distributed telecommunication software [46] [47]. The wide ranges of use of MSCs are usually in the distributed systems and in a number of software methodologies [47] [48] [49]. In a distributed system, MSCs mainly concentrate on the exchange of messages among various processes and their environments as well as some internal actions in these processes. MSCs are also known as object interaction diagrams, timing sequence diagrams and message flow diagrams.

In MSCs, the executing processes are shown by the vertical lines; these processes communicate through an explicit message passing (send-receive) among them shown by the horizontal or downward sloped arrow lines. The head of the arrow indicates to the event *message-receiving* and the opposite end indicates to the event *message-sending*. Each *send-receive* event (horizontal or downward sloped line) is labeled by the message identifier. For more clear understanding the MSC may also contain necessary data attributes as part of the message exchanged. A simple MSC is shown in the following Fig. 4 where there are two processes namely 'CPU' and 'Memory'.

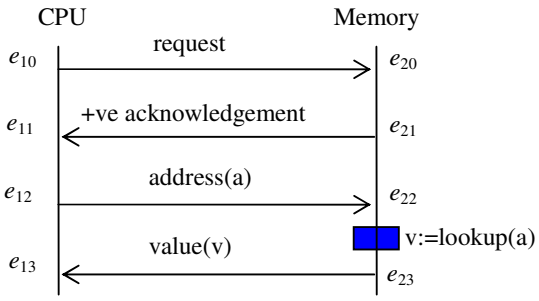


Fig. 4: A Message Sequence Chart

Time flows downward in each vertical line of MSCs. So, in this MSC, the sequences of actions in process 'CPU' and 'Memory' are {sending request, receiving +ve acknowledgement, sending address *a* and receiving value of address *a*} and {receiving request, sending +ve acknowledgement, receiving address *a*, an internal action $v : lookup(a)$ and sending value *v*} respectively. These orderings cannot be violated in either of the processes i.e. a total ordering of the events along every process is assumed. Every process of the MSC is assumed to contain a message queue to store the incoming messages and another message queue to store the outgoing messages. Each MSC is associated with a 'condition'. The MSC is executed when its 'condition' is *true*. The 'condition' is composed by predicates over the variables of the processes in the MSC connected by the logical connectives. The 'condition' of the above MSC is $CPU.status \wedge Memory.ready$. The variable *status* is a boolean variable of the process *CPU* which is true when the *CPU* is ready to interact with *Memory*. Similarly, the variable *ready* is a boolean variable of the process *Memory* which is true when it can serve the request of the *CPU*.

Let us discuss MSCs with its formal definition. Assume that P is the finite set of processes, M is the finite set of messages and A is the finite set of internal actions. For each $p \in P$, a set of events the process p takes part in is defined by $\sum_p = \{< p!q, m > | p \neq q, q \in P, m \in M\}$ $\cup \{< p?q, m > | p \neq q, q \in P, m \in M\}$ $\cup \{< p, a > | a \in A\}$. The meanings of $< p!q, m >$, $< p?q, m >$ and $< p, a >$ are 'process p sends message m to process q', 'process p receives message m from process q' and 'process p performs internal action a' respectively. We set $\sum = \cup_{p \in P} \sum_p$ and let α, β range over $\sum$. Assume a set of channel $Ch = \{(p, q) | p \neq q\}$ and let c, d range over Ch. A $\sum$-labeled poset is a structure $S = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \sum$ is a labeling function. Here E is a finite set of events and $\leq$ is a partial order which is *reflexive*, *transitive* and *anti-symmetric*. For any event $e \in E, \downarrow e = \{e_1 | e_1 \leq e\}$ where $e_1 \leq e$ means that the event e_1 occurs before event e. For $p \in P$, and $a \in \sum$, let $E_p = \{e | \lambda(e) \in \sum_p\}$ and $E_a = \{e | \lambda(e) \in a\}$. For channel c, let the relation $R_c = \{(e, e_1) | \lambda(e) = p!q, \lambda(e_1) = p?q$ and $|\downarrow e \cap E_{p!q}| = |\downarrow e_1 \cap E_{p?q}|\}$. For a process $p \in P$, the relation is $R_p = (E_p \times E_p) \cap \leq$ The R_{ch}-edge across the processes is depicted by the horizontal or downward sloped edge.

Thus, an MSC (over P) is a finite $\sum$-labeled poset $S = (E, \leq, \lambda)$ that satisfies the following conditions [50]:

a) For every $p \in P$, R_p is a linear order.

b) For every $\{p, q\} \in P$ and $p \neq q$, $|E_{p!q}| = |E_{q?p}|$ i.e. no lifeless communication edge exists in MSC that means the number of *sent* messages equals the number of *received* messages.

c) $\leq = (R_p \cup R_{Ch})^*$ where $R_p = \cup_{p \in P} R_p$ and $R_{Ch} = \cup_{c \in Ch} R_c$ i.e. the partial order of MSC is its visual order; deduced by linear orders of participating processes and the *sent-receive* order of the messages.

***Modeling the Communications*:**

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, we model it as a queue of message, which is the the integral part of the threads. The following Fig. 5 shows the modeling of the channel as a queue of message from thread *i* to thread *i*+1. The message is pushed through the *tail* of the queue from the thread *i* side and the message is received from the *head* of the queue at the thread *i*+1 side. Fig. 6 shows the modeling of the communication channel as a queue for the message from thread *i*+1 to thread *i*. The message is sent from the thread *i*+1 side and it is received at the *head* of the queue at the thread *i* side.

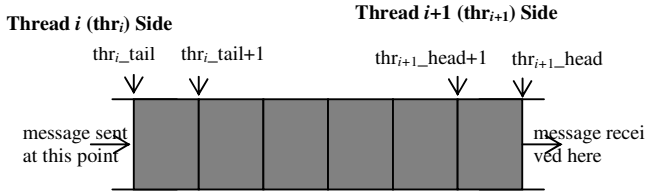


Fig. 5 Channel for message from thread *i* to thread *i*+1

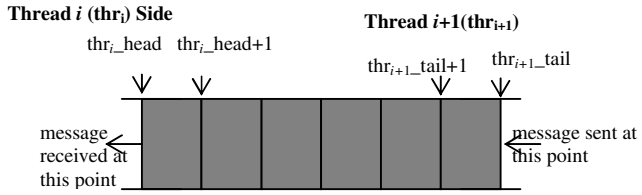


Fig. 6 Channel for reply from thread *i*+1 to thread *i*

The communication among threads is specified by a grammar. The syntax for it is formulated and is tabulated in Table 1.

Table 1: The Syntax

S → **THREAD thread-name {** thr-dec-part trans-sch-dec-part **}**
thr-dec-part → thr-dec-part thr-dec | thr-dec
thr-dec → **PROCESS process-name {**var-dec-part equation-part **}**
type → basic-type | enum-type | array-type
basic-type → range-type | **BOOLEAN**
enum-type → **{** id-list **}**
id-list → **identifier (, identifier)**$^{+}$
array-type → **ARRAY** range-type **OF** basic-type
range-type → **interger-const .. interger-const**
var-dec-part → var-dec-part var-dec**;** | ε
var-dec → **type-id :** id-list
equation-part → **EQUATION** id-list**;**
trans-sch-dec-part → trans-sch-dec trans-sch-dec-part | trans-sch-dec
trans-sch-dec → **SCHEME trans-schm-name {** trans-list **}**
trans-list → trans-dec trans-list | trans-dec
trans-dec → **TRANSACTION trans-name {**
AGENTS { agent-list **}** condition-section **; }**
| TRANSACTION trans-name { AGENTS { agent-list **}**
condition-section → **CONDITION** condition **;**
agent-list → agent-list agent | agent
agent → **process-name :** event-list
event-list → event **,** event-list | event **;**
event → **send(mesg-id, var) | send(mesg-id, const, type) | recv(process-name.mesg-id) |** {action}
action-atom → simple-stmt **;** | if-stmt
action → action action-atom | action-atom
simple-stmt → **var :=** expr | **var :=DIN**
expr → expr [* | / | &] F | F
F → F **+** G | F **-** G | F | G
G → G **mod** H | H
H → H **RelOp** I | I
I → ~I | -I | (expr) | var | const
const → **integer-const | boolean-const**
condition → condition & condition-atom | condition-atom | (condition-atom)
condition-atom → ~prop | prop
prop → prop **or** prop-atom | prop-atom
prop-atom → scoped-var relop const | scoped-var relop scoped-var | scoped-var
relop → = | < | > | ≤ | ≥ | !=
if-stmt → **IF** expr **{** action **} | IF** expr action-atom **| IF** expr **{**action**}ELSE {**action**}**
var → **identifier | identifier[identifier] | identifier [integer-const]**
scoped-var → **process-name.**var
thread-name → **identifier**
process-name → **identifier**
trans-name → **identifier**
mesg-id → **identifier**

B. Model Verification

Let us first be familiar with transition system and Kripke Structure before we define *model*. A transition system is a structure $TS = (S, S_0, R)$ where, S is a finite set of states; $S_0 \subseteq S$ is the set of initial states and $R \subseteq S \times S$ is a transition relation which must be total i.e. for every s in S there exists s_1 in S such that (s, s_1) is in R $\left(\forall_{s \in S} \exists_{s_1 \in S}, (s, s_1) \in R\right)$. And $M = (S, S_0, R, AP, L)$ is a Kripke Structure; where (S, S_0, 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 [51]. 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* $K = (S, S_0, R, AP, L)$ (system model) and a CTL formula Φ (specification of the property) are given. We have to determine if $K \models \Phi$ holds (K is a model of Φ) or not. $K \models \Phi$ holds iff $K, s_0 \models \Phi$ for every $s_0 \in S_0$. 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*** Ξ (reads 'Ξ' holds sometime in the future") is true in a path if there exists a state in that path where formula 'Ξ' is true.
- ***G*** Ξ (reads 'Ξ' holds globally") is true in a path if 'Ξ' is true at each and every state in that path.
- ***X*** Ξ (reads 'Ξ' holds in the next state") is true in a path if 'Ξ' is true in the state reached immediately after the current state in that path.
- Ξ ***U*** Ω (reads 'Ξ' holds until 'Ω' holds) is true in a path if 'Ω' is true in some state in that path, and 'Ξ' holds in all preceding states.

The semantics of the CTL operators are below:

- $K, s \models EX(\Phi)$ there exists s_1 such that $s \to s_1$ ($R(s, s_1)$) and $K, s_1 \models \Phi$. It means that s has a successor state s_1 s' at which Φ holds.
- $K, s \models EU(\Phi_1, \Phi_2)$ iff there exists a path $L = s_0, s_1 \ldots\ldots$ from s and $k \geq 0$ such that: $K, L(k) \models \Phi_2$ and if $0 \leq j < k$, so $K, L(j) \models \Phi_1$.
- $K, s \models AU(\Phi_1, \Phi_2)$ iff for every path $L = s_0, s_1 \ldots\ldots$ from s there exist $k \geq 0$ such that: $K, L(k) \models \Phi_2$ and if $0 \leq j < k$, so $K, L(j) \models \Phi_1$.
- $AX(\Phi)$: It is not the case there exists a next state at which Φ does not hold i.e. for every next state Φ holds.
- $EF(\Phi)$: There exists a path L from s and $k \geq 0$ such that: $K, L(k) \models \Phi$.
- $AG(\Phi)$: It is not the case there exists a path L from s and $k \geq 0$ such that: $K, L(k) \models \Phi$ i.e. for every path L from s and every $k \geq 0$; $K, L(k) \models \Phi$.
- $AF(\Phi)$: For every path L from s, there exists $k \geq 0$ such that: $K, L(k) \models \Phi$.
- $EG(\Phi)$: It is not the case that for every path L from s there is a $k \geq 0$ such that $K, L(k) \models \Phi$. It means that there exists a path L from s such that, for every $k \geq 0$: $K, L(k) \models \Phi$.

C. Verification

Despite tremendous advances in constraint-random stimulus generation and coverage-based verification, simulation-based verification cannot effectively find all the potential issues within today's complex designs. This is the main reason that designers complement their simulation-based verification methodology with formal verification.

We use the SMV [52] as the verification tool. Here the number of threads is defined as SIZE = 8. Using the loop from 0 to 7 we verify if any of the threads starts vertical transformation while some other thread(s) is/are still on the transformation of the horizontal part of the same section (unsafe). It is found that each of the threads is in the safe state (desired) all the time. It means that in all states of the transition system it is true that no vertical transformation gets started until the all the horizontal section is completed. This specification is the most important one that we must get true. The part of the SMV code is shown in Fig. 7 and the snapshot of the verification result is shown in Fig. 8.

```
for (index=0;index<SIZE;index=index+1){
   loc[index],idle[index]:boolean;
   idle[index]:=free & freed=index;
   loc[index]:=alloc & pos_ack & allocd=index;
   safe[index]:SPEC AG(A(~idle[index] U loc[index]));
}
```

Fig.7 Part of the SMV Code

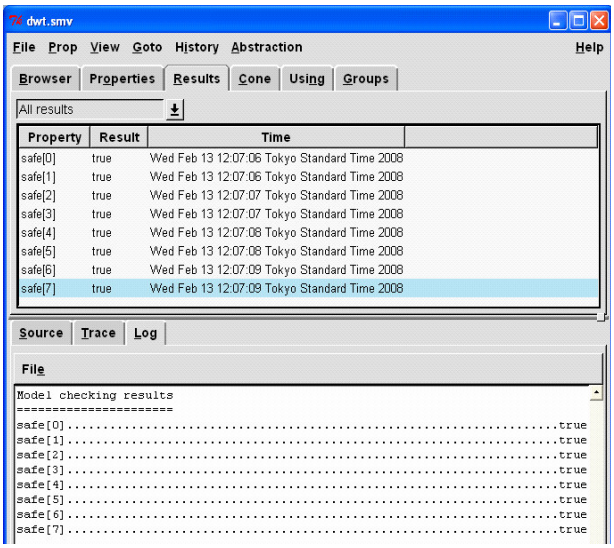


Fig. 8 Verification Results

V. Conclusion

The concept of compression, the use of wavelet for the image compression, the possibility of concurrency in the wavelet transformation with a model, the communication among different concurrent threads for wavelet transformation and the verification result of the model are presented here. The system is modeled in SMV and verified by SMV that automatically creates a formal environment to efficiently solve the design checking tasks. One of the important properties, in the context of the concurrent wavelet transformation, is that at any time, the vertical transformation does not start on a vertical section that is not fully horizontally transformed. We find that all the communicating threads are in the *safe* states all the time. At present, the system has two drawbacks. One is, in SMV, the smaller size of the queue shared by different threads and the other is the smaller number of participating threads. In this respect, we plan to model it in SPIN, another verification tool, in future, to solve these problems.

References

[1] R. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, 1991.

[2] C. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:623-656, 1948.

[3] N. Faller. An adaptive system for data compression. In *Proceedings of the Asilomar Conference on Circuits, Systems and Computers*, pages 593-597, 1973.

[4] R. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24:668-674, 1978.

[5] D. Knuth. Optimal binary search trees. *Acta Informatica*, 1:14-25, 1971.

[6] J. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the Association for Computing Machinary*, 34:825-845, 1987.

[7] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337-343, 1977.

[8] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530-536, 1978.

[9] T. Huang. Run length coding and its extensions. In T. Huang and O. Tretiak, editors, *Picture Bandwidth Compression*, pages 231-264. Gordon and Breach, 1972.

[10] A. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

[11] J. Lim. *Two-Dimensional Signal and Image Processing*. Prentice-Hall, 1990.

[12] D. O'Shaughnessy. *Speech Communication: Human and Machine*. Addison-Wesley, 1987.

[13] M. Vetterli. Multi-dimensional sub-band coding: Some theory and algorithms. *Signal Processing*, 6:97-112, 1984.

[14] J. Woods and S. O'Niel. Subband coding of images. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34:1278-1288, 1986.

[15] J. Lim. *Two-Dimensional Signal and Image Processing*. Prentice-Hall, 1990.

[16] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28:84-95, 1980.

[17] J. Makhoul, S. Roucos, and H. Gish. Vector quantization in speech coding. *Proceedings of the IEEE*, 73:1551-1558, 1985.

[18] J. Abate. Linear adaptive delta modulation. *Proceedings of the IEEE*, 55:298-308, 1967.

[19] S. Alexander and S. Rajala. Image compression results using LMS adaptive algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 33:712-717, 1985.

[20] C. Cutler. Differential quantization of communication signals. U.S. Patent 2 605 361, 1952.

[21] A. Habibi. Survey of adaptive image coding techniques. *IEEE Transactions on Communications*, 25:1275-1284, 1977.

[22] N. Jayant. Adaptive delta modulation with a one-bit memory. *Bell Systems Technical Journal*, 49:321-343, 1970.

[23] A. Kolmogorov. Interpolation and extrapolation of stationary random series. *Journal of the Soviet Academy of Science*, pages 3-14, 1941.

[24] T. Lei, N. Scheinberg, and D. Schilling. Adaptive delta modulation system for video encoding. *IEEE Transactions on Communications*, 25:1302-1314, 1977.

[25] J. O'Neal. Predictive quantization system (differential pulse code modulation) for the transmission of television signals. *Bell Systems Technical Journal*, 45:689-721, 1966.

[26] P. Pirsch. Adaptive intra/interframe DPCM coder. *Bell Systems Technical Journal*, 61:747-764, 1982.

[27] C. Song, J. Garondick, and D. Schilling. A variable-step-size robust delta modulator. *IEEE Transactions on Communications*, 19:1033-1044, 1971.

[28] N. Ahmed, T. Natarajan, and K. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, 23:90-93, 1974.

[29] G. Anderson and T. Huang. Piecewise Fourier transformation for picture bandwidth compression. *IEEE Transactions on Communications*, 19:133-140, 1971.

[30] B. Fino. Relations between Haar and Walsh/Hadamard transforms. *Proceedings of the IEEE*, 60:647-648, 1972.

[31] H. Andrews. *Computer Techniques in Image Processing*. Academic Press, 1970.

[32] A. Mazzarri and R. Leonardi. Perceptual embedded image coding using wavelet transforms. In *Proceedings of the International Conference on Image Processing*, volume I, pages 586-587, 1995.

[33] Gortler. S., Schröder, P., Cohen, M., and Hanrahan, P., "Wavelet Radiosity", in Proc. SIGGRAPH, 1993, pp. 221-230.

[34] Berman, D., Bartell, J. and Salesin, D., "Multiresolution Painting and Compositing", in Proc. SIGGRAPH, 1994, pp. 85-90.

[35] Finkelstein. A. and Salesin, D., "Multiresolution Curves", in Proc. SIGGRAPH, 1994, pp. 261-268.

[36] 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.

[37] Lippert, L. and Gross, M., "Fast Wavelet Based Volume Rendering by Accumulation of Transparent Texture Maps", in Proc. EUROGRAPHICS, 1995, pp. 431-443.

[38] Jacobs, C., Finkelstein, A. and Salesin, D., "Fast Multiresolution Image Querying", in Proc. SIGGRAPH, 1995, pp. 277-286.

[39] Myers, Glenford J. "The Art of Software Testing", John Wiley and Sons. ISBN 0-471-04328-1, 1979.

[40] Edmund M. Clakre, Jr. Oma FrumBerg and Doron A. Paled, "Model Checking", The MIT Press, Second Printing, 2000.

[41] Eric J. Stollnitz, Tony D. Derose and David H. Salesin, "Wavelets for Computer Graphics", Morgan Kaufmann Publishers, Inc., San Francisco.

[42] S. Mallat, "A theory of multiresolution signal decomposition: The wavelet representation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 11, pp. 674–693, July 1989.

[43] S. Mallat, "Multifrequency channel decomposition of images and wavelet models," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 2091-2110, 1989.

[44] S. Mallat, "Wavelets for a vision" *Proc. IEEE*, vol. 84, pp. 604–614, 1996.

[45] Ekkart Rudolph, Peter Graubmann and Jens Grabowski: *Tutorial on Message Sequence Charts.* MSC Tutorial of the 7th SDL Forum, September 1995, Norway.

[46] ITU-TS Recommendation Z.120: *Message Sequence Chart (MSC).* ITU-TS, Geneva (1997).

[47] Rudolph, E., Graubmann, P. and Grabowski, J.: *Tutorial on Message Sequence Charts*. In Computer Networks and ISDN Systems-SDL and MSC, Volume 28 (1996).

[48] Booch, G., Jacobson, I. and Rumbaugh, J.,: *Unified Modeling Language User Guide*. Addison-Wesley (1997).

[49] Harel, D. and Gery, E.: *Executable object modeling with statecharts*. IEEE Computers, July 1997, pp. 31-42.

[50] Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar and P.S. Thiagarajan: *Regular Collections of Message Sequence Charts*. Published in Springer Lecture Notes in Computer Science, pp. 405-414. In Mathematical Foundation of Computer Science (MFCS).

[51] 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.

[52] Cadence Berkeley Laboratories, Free download from the website: http://www.kenmcmil.com/smv.html, Califonia, USA. SMV Model Checker, 1999.