International Journal of Computer and Communication Technology

Volume 1 | Issue 1 Article 8

January 2010

Precise Descriptions of VLC Synchronization with CSP Semantic Models

A.C.M . Fong
School of Computing and Mathematical Sciences, Auckland University of Technology, New Zealand., acmfong@gmail.com

Precise Descriptions of VLC Synchronization with CSP Semantic Models

A.C.M. Fong*
School of Computing and Mathematical Sciences,
Auckland University of Technology, New Zealand.
Email: acmfong@gmail.com
*Corresponding Author

Andrew Simpson
Oxford University Computing Laboratory,
England

Bernard Fong
Centre for Prognostics and Health Management
City University of Hong Kong,
Hong Kong

Abstract: Variable length codes (VLC) have found widespread applications due to their inherent coding efficiency. However, encoder-decoder synchronization becomes critically important for VLC to operate properly. Traditional tree-based techniques lack the scalability to analyse the synchronization behaviours of VLC, and simulation techniques are typically used instead for large code sets. Building on an initial paper in which we first described an application of CSP to this domain, we present further advances in this paper. The contributions of this paper are twofold. First, we describe a novel application of the CSP stable failure model to completely describe the VLC synchronization mechanisms. Consequently, we concisely characterize bit patterns that can bring about rapid synchronization. The overall goal is to advance our understanding in this important area of research through an established formal description technique originally developed and used within the computing research community.

Keywords: CSP, formal description techniques, VLC synchronization

Reference to this paper should be made as follows: Fong, A.C.M. *et al.* (2009) 'Precise descriptions of VLC synchronization with CSP semantic models', *Int. J.CCT*, Vol.1, No.1, pp.59-70.

Biographical notes: A.C.M. Fong is Professor of Computer Engineering in the School of Computing and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand. He was educated at Imperial College London, and the Universities of Oxford and Auckland. His research interests include communications, software engineering, and internet and multimedia technology. He is Editor-in-Chief of the Journal of Advances in IT. Dr. Fong is a senior member of the IEEE and a Chartered Engineer registered in the UK.

Andrew Simpson is a University Lecturer at the Oxford University Computing Laboratory. His main research interests can be classified into two broad categories: the design and development of middleware to support the secure sharing and aggregation of data from disparate sources, and the development of tools and techniques to support assured, context-sensitive access control.

Bernard Fong is a Senior Research Fellow of CityU Centre for Prognostics and Health Management at the City University. He received his BS degree from University of Manchester Institute of Science and Technology, United Kingdom, and PhD in healthcare information systems from the University of New South Wales, Australia.

1 Introduction

Variable length codes (VLC) e.g. (Huffman, 1952) (Titchener, 1997) (Zhou and Zhang, 2002) have found widespread applications in multimedia and communication systems for message encoding e.g. (Fong and Quay, 2000) and (Fong *et al*., 2001) due to their inherent coding efficiency compared to fixed length codes (FLC) such as ASCII. In this context, messages can come from images (e.g. run-length codes in a baseline JPEG coding scheme), text documents, etc. The fundamental requirement is that the messages should be encoded with the minimum number of bits without jeopardizing subsequent decoding and recovery.

Encoder-decoder synchronization becomes critically important for VLC to operate properly. With FLC, it is relatively easy for the decoder to ascertain the codeword boundaries precisely because each codeword has the same fixed length. However, the coding efficiency of FLC cannot match that of well-constructed VLC. In general, we require that practical VLC will attain resynchronization in finite time following a lock loss. VLC that possess this property are considered statistically synchronizable (Capocelli *et al*., 1992). Clearly, VLC that do not provide even this level of guarantee are of little practical significance. Of particular interest are VLC that exhibit a strong tendency towards automatic synchronization following a lock loss. VLC that have this property are very useful in limiting error propagation. These VLC are typically referred to as self-synchronizing VLC e.g. (Ferguson and Rabinowitz, 1984) (Takishima *et al*., 1994) (Titchener, 1997).

However, it requires a thorough understanding of the complex nature of the synchronization mechanisms of VLC to fully realize their potential. This would be essential for finding (or constructing) universally good code sets for different information sources. Unfortunately, this remains an open research issue. Some researchers even suggest that general understanding of the mechanisms of self-synchronization may be an unattainable goal e.g. (Titchener, 1997) and (Zhou and Zhang, 2002).

Traditionally tree-based techniques e.g. (Takishima *et al*., 1994) and (Fong and Higgie, 2002) are used to analytically study the synchronization behaviours of VLC with the aim of ultimately finding ways to construct universally good codes that are both efficient and possess strong self-synchronizing properties. However, these techniques lack the scalability to cover large code sets, and simulation techniques are typically used instead (Higgie, 1996). Communicating Sequential Processes (CSP) (Hoare, 1985) is established process algebra for describing the patterns of communication and interaction between agents that interact via explicit message passing. CSP has been developed and used by members of the software community for years. We have found that some aspects of CSP seem ideally suited to the description of VLC synchronization. In (Fong and Simpson, 2006), we presented an exploratory study of applying CSP to the modelling of VLC synchronization. In that paper, we described VLC synchronization as mutual recursions using the CSP external choice operation.

Building on that initial work, we present further advances in this paper. The contributions of this paper are twofold. First, we apply the CSP stable failure model to completely describe the VLC synchronization mechanisms. Consequently, we can concisely characterize bit patterns that can bring about rapid synchronization. Known as synchronizing sequences, these bit patterns can exist naturally in the bit stream arriving at the decoder. These sequences appear most frequently with well-constructed VLC that exhibit strong tendency towards automatic synchronization. Their relative frequencies of occurrence in a bit stream can therefore give a measure of how different code sets perform in terms of automatic synchronization.

The rest of this paper is organized as follows. In Section 2 we present a brief summary of the relevant aspects of CSP to facilitate further discussion. Section 3, which is the main section of this paper, illustrates with the aid of some examples how to precisely describe the VLC decoding/synchronization mechanism using the CSP stable failure model. Section 4 then demonstrates an application of the stable failure descriptions by deducing synchronizing sequences. Finally, Section 5 concludes the paper.

2 Related work on CSP

This section presents the basics of CSP such as the processes and events, prefix choice, determinism, recursions, etc., as well as the traces and stable failure models. Other CSP models are not required. For example, given that all VLC in this study are statistically synchronizable as defined in Section 1, infinite bit sequences are outside the scope of this study.

2.1 Basic CSP

CSP (Hoare, 2985) is an established formal method for modelling complex, concurrent processes. The fundamental elements of a CSP model are events and processes. A process interacts with its environment and other processes via its interface. The interface (or alphabet) of a process consists of a set of events. The process may then offer to engage in some activity in response to an event, though there is no guarantee that any of the allowable events occur. A complete model made up of individual processes and events can be constructed to describe complex systems. The use of processes and events as building blocks makes this a highly scalable approach.

There are fundamentally four untimed semantic models of CSP: Traces, Stable Failures, Failures-Divergences, and Failures, Divergences and Infinite Traces (FDI). These semantic models provide different levels of abstraction to the system analyst. A trace is a record of observable (external) events that have taken place, but does not contain information on other allowable events that could have taken place. Also, the traces model does not model divergence. Divergence occurs when a process is permitted to execute internal events to such an extent that the process no longer engages in any external events being offered to it. Once a process enters into a divergent state, its behavior becomes unstable. A stable process P is denoted by $P\downarrow$. On the other hand, if P is unstable, it is denoted by $P\uparrow$.

While a trace records the events that have occurred up to any point during process execution (including the initial state when the trace is empty), a refusal is a set of events that can be refused at the same point. Together, they form a failure.

As the name implies, the FDI model allows one to study processes that can diverge, as well as have infinite traces. The consideration of only statistically synchronizable codes eliminates the need to consider divergence and infinite traces in the CSP models. Thus, only the Traces and Stable Failures models are relevant. The details of CSP can be found in references such as (Hoare, 1985) and (Roscoe, 1997) and will not be reproduced here. Instead, the following discussion highlights the aspects of CSP that are relevant to this study.

Process. A process P is a fundamental, self-contained building block of a CSP system that has interfaces through which it can communicate with the outside world, including other processes in the same environment. It is through this composition of communicating processes that makes CSP so useful for reasoning about large systems made up of many processes that interact with each other. Consequently, CSP models developed for relatively small systems (with relatively few processes) can be scaled up quite easily.

Alphabet. An alphabet □P defines the interface of a process P. In the present context, the main focus is on binary sequences. At a bit level, for example, the alphabet of a process NODE in any binary tree may be the set {zero, one}, which suggests that the process can accept or output zeros and ones, but do nothing else.

Predefined CSP processes. There are several of these. However, the ones that are of interest to us include STOP (deadlock) and SKIP (successful termination). The special termination event that brings about a successful termination is denoted by $\surd$.

Set of events. Denoted by $\sum$, this is the set of possible external (observable and controllable) events within a particular specification (allowable interface events). The union of $\sum$ and $\surd$ is abbreviated $\sum^{\surd}$.

Set Subtraction. If the set Error States contains the elements ES_1, ES_2 and ES_3, then Error States\ $\{ES_2\} = \{ES_1, ES_3\}$.

Event Prefix and Prefix Choice. If $a \in \sum$, then the construct $a \rightarrow P$ describes a process that can initially engage in the event a and thereafter behaves as process P. Prefix choice is a generalization of this. If A is a set of events and the process P is defined for all members of A, then $x : A \rightarrow P(x)$ means that the prefix is an event chosen from the set A. This causes the process P to behave in response to this chosen value of x in A.

External Choice and Internal Choice. The process $P_1 = a \rightarrow P_2 \,\square\, b \rightarrow P_3$ means the process P_1 is prepared to engage in event a or event b, and will then behave as P_2 if a was chosen or P_3 if b was chosen. The choice is made externally by the environment, which means it is possible for an observer to control the outcome by presenting the event of choice to P_1. Obviously, $\alpha\ (P_1 \square P_2) = \alpha\ (P_1) \cup \alpha\ (P_2)$.

The process $Q_1 = a \rightarrow Q_2 \prod b \rightarrow Q_3$ means that it may refuse event a or event b, but it may not refuse both. It will then behave as Q_2 if a was chosen or as Q_3 if b was chosen. The process makes the choice internally so that the environment has no influence on the choice. An internal transition is denoted by $\square$. Obviously, $\alpha\ (Q_1 \prod Q_2) = \alpha\ (Q_1) \cup \alpha\ (Q_2)$.

The external and internal choices can be generalized as indexed choices as $\square_{i \in I} P_i$ and $\prod_{i \in I} Q_i$, and their respective alphabets are $U_{i \in I} \square(P_i)$ and $U_{i \in I} \square(Q_i)$.

Determinism. The notion of determinism is closely related to whether or not a process can engage in internal transitions. If a process can engage in internal transitions (where an event cannot be recorded by an observer), it may no longer interact with its environment. In this case, it becomes non-deterministic. So, the external (internal) choice is also known as deterministic (non-deterministic) choice.

Recursion and Mutual Recursion. The process SYNC = zero $\rightarrow$ one $\rightarrow$ SYNC describes the events zero, one performed in an infinite loop. This same behavior can also be described as a mutual recursion by introducing an intermediate process.

Sequential Composition. The sequential composition of two processes P_1 and P_2, denoted by P_1 ; P_2, behaves as P_1 initially until it terminates successfully. Thereafter, the composite process behaves as P_2.

Concurrent Processes. In the present study, there are two relevant operations through which two or more concurrent CSP processes can co-exist in an environment, namely the parallel and interleaving operations. In the case of parallel processes, interaction takes place by synchronizing on the events in the interface of the processes. The notation for describing two parallel processes P_1 and P_2 is $P_1 \parallel P_2$. The alphabet of the combined process is $\alpha\ (P_1 \parallel P_2) = \alpha\ (P_1) \cup \alpha\ (P_2)$.

It is possible to describe multi-way event synchronization involving a composition of multiple parallel processes. For example, one can describe parallel processes such as $P_1 \parallel P_2 \parallel P_3$, or $(P_1 \parallel P_2) \parallel P_3$. The parallel operation can be generalized as the indexed parallel of the form $\parallel^{i \in I} P_i$. The alphabet of the indexed parallel is $\square(\parallel^{i \in I} P_i.) = U_{i \in I} \square(P_i)$.

Deadlock can arise when parallel processes cannot agree on the events to synchronize such that no progress can be made when each participating process is waiting for an impossible event to occur. In some situations, there is no need for concurrent processes to synchronize on certain events. For example, it is possible that the processes Q_1 and Q_2 can operate independently without any need for interaction, except that they terminate simultaneously. This can be described by the interleaving operation using the notation $Q_1 ||| Q_2$. The alphabet of the combined process is $\alpha\ (Q_1 ||| Q_2) = \alpha\ (Q_1) \cup \alpha\ (Q_2)$.

A special case occurs when two parallel processes P_1 and P_2 do not have any event in common, such that there is no synchronization between them. In this case, $P_1 \parallel P_2$ behaves as $P_1 ||| P_2$ because the two processes just go about their own business without ever needing to agree on any common progress. Again, it is possible to combine multiple interleaving processes, e.g. $Q_1 ||| Q_2 ||| Q_3$ and $(Q_1 ||| Q_2) ||| Q_3$. The interleaving operation can also be generalized as the indexed interleaving operation denoted by $|||^{i \in I} Q_i$. The alphabet of this is $\square(|||^{i \in I} Q_i.) = U_{i \in I} \square(Q_i)$.

2.2 The CSP traces model and stable failure models

Traces. A trace is a record of external events visible to an observer of an execution of a process. For example, a possible trace of the process SYNC is ⟨zero, one, zero⟩. A trace can also be empty⟨⟩, which means no allowable event has taken place. In addition, the notation #(tr) gives the length of the trace tr , which could give a quantitative measure of the synchronization performance in terms of the number of bits or code words required to reestablish synchronization following a lock loss (i.e. a measure of synchronization delay).

Stable Failures. By considering only statistically synchronizable code sets, the possibilities of having divergence and infinite traces are ruled out. Stable failures concern the analysis of the liveness property of a process. First, a process P must be stable (non-divergent). From the discussion in Section 2.1 above, this means P cannot enter into an infinite sequence of internal transitions.
For example, the stable failure of the process $a \rightarrow P$, denoted by $SF[a \rightarrow P]$ is the set $\{(\langle\rangle, X) \mid a \notin X\}$ U $\{(\langle a\rangle \,^\wedge\, tr, X) \mid (tr, X) \in SF[P]\}$, where X is the refusal and tr represents the trace after a has occurred. The stable failure is therefore a union of two sets as follows. The first set states that the event a has not occurred, so the trace is empty and the process is able to refuse any event except a. The second set means that the event a has occurred and the rest of the stable failures is determined by P. Thus, stable failures give a view of which events will be refused (not able to proceed) by a process at any stage of execution. This in turn facilitates checking of the liveness behavior of a process at any stage of execution.

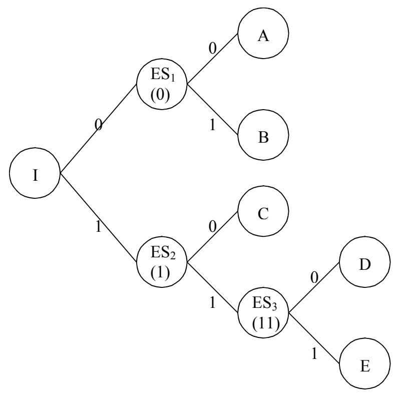
Failures can also be used to check for determinism. For example, if $\sum = \{a, b\}$. Then, the process $P = a \rightarrow STOP \;\square\; b \rightarrow STOP$ initially cannot refuse either event a or event b (refusal set $X = \{\}$), but once any of these events has occurred, both events can be refused. So, the failures of P are $\{(\langle\rangle,\{\})\}$ U $\{(\langle a\rangle, X) \mid X \subseteq \{a, b\}\}$ U $\{(\langle b\rangle, X) \mid X \subseteq \{a, b\}\}$.

On the other hand, the process $P = a \rightarrow STOP \prod b \rightarrow STOP$ is initially unstable due to the internal choice. There is no way of influencing the resolution of the choice. In either case, $Q \rightarrow^{\square} Q_1$, where Q_1 is now stable. Once Q_1 is reached, either event a or event b is possible and the other is refused. So, $X = \{a\}$ or $\{b\}$ at this stage. Once any one of these events occurs, both events can be refused. So, the failure of Q is a union of three sets as follows: $\{(\langle\rangle,\{\}), (\langle\rangle,\{a\}), (\langle\rangle,\{b\})\}$ U $\{(\langle a\rangle, X) \mid X \subseteq \{a, b\}\}$ U $\{(\langle b\rangle, X) \mid X \subseteq \{a, b\}\}$. Notably, the first set distinguishes the internal choice from external choice.

3 Stable failure model for VLC synchronization

We now explore the applicability of the CSP stable failure model to the decoding and synchronization of VLC. Intuitively, the set of refusals at any stage of the decoding process is conceptually similar to blocking conditions that have an accumulative effect on the delay to the resynchronization process following a lock loss (Recall that resynchronization amounts to finding the location of the next codeword boundary in the bi stream, a task performed by the decoder during the decoding process). We exploit this similarity to model the blocking conditions encountered during decoding between the point of lock loss and the point of resynchronization.

Figure 1 Decoding tree of code set C_{01}



In practice, we require all VLC to be exhaustive. This means the corresponding encoding/decoding tree is fully populated. In view of this, the CSP stable failure model is not really applicable to the “atomic” approach of considering individual bits as events (as in Model A described in (Fong and Simpson, 2006)). A simple counterexample illustrates this point. Figure 1 shows the binary decoding tree for a simple code set known as C_{01} in (Takishima *et al*., 1994).

In Figure 1, we observe that at any stage of decoding, the ES_i cannot refuse either 0 or 1. So, the refusal set would always be empty. Thus, the traces model is most suitable when one takes the atomic approach. In particular, by recording the trace of the decoding processing from process I to process S, the length of this trace gives the number of bits required to resynchronize, which gives a measure of synchronization delay. Thus, the search for optimal codes can be restricted to those that give the minimum values of length of the trace among the code sets in the same subgroup.

On the other hand, the concept of sets of refusals is very much applicable to models that consider valid code words or other special bit sequences as events (as in Model B described in (Fong and Simpson, 2006)). In this context, possible events include prefixes and suffixes of valid code words. To facilitate referencing, the prefixes and suffixes of valid code words, together with the code words themselves, are collectively known as \valid" bit sequences. Intuitively, the bit stream that arrives at the decoder at any stage of decoding can be either the suffix of a final-level valid codeword (which allows progression through to the next stage of decoding) or some other concatenation of bits (which may cause blockage that impedes further progress in the decoding process causing additional delay to resynchronize).

We consider each level of decoding as a process P_i, and to look at how decoding progresses from one process (level) to the next sequential process (next higher level). Recall that decoding begins from level 0 immediately following a lock loss, and progresses through intermediate levels (1, 2, …) until the final level n is reached and synchronization is reestablished. So, it is clear that i ranges from level 0 to level n. The next step is to apply sequential composition to the individual processes P_i so as to ensure that P_i terminates successfully before progressing to P_{i+1}. This means we insist that a successful decode in level i causes decoding of the following bits to continue in the next level.

The stable failure of the sequential composition of two processes P_1 and P_2 is $\{(tr, X) \mid (tr, X \cup \{\surd\}) \in SF[P_1]\} \cup \{(tr_1 \,\hat{}\, tr_2, X) \mid (tr_1 \,\hat{}\, \langle\surd\rangle \in traces(P_1) \wedge (tr_2, X) \in SF[P_2]\}$. This means the stable failure of the composite process is either from a failure of P1 which has not terminated, or from a terminating trace of P_1 and a failure of P_2. This argument can be applied to a sequential composition of multiple processes, each of which represents decoding in a level. The moment when the final sequential process terminates signals the reestablishment of synchronization, and proper decoding resumes at that point. The following example illustrates the proposed method.

Example: Consider the augmentation level 4 T-code[1] set $S_{0,1,00,101}^{1,1,1,1}$, which is now referred to as C_{02} and was also used as an illustration in (Fong and Higgie, 2002) for tree construction analysis. Table 1 highlights the important part of the code construction relevant to our study. C_{02} is chosen for illustration here because the decoding process involves both prefix and suffix blocking conditions, both of which contribute towards synchronization delay and are to be described as refusals.

The prefix blocking condition occurs when decoding in a level L_i if the decoded codeword CW is exactly the same as prefix for L_{i+1}. This is because the CW is "sacrificed" and augmented to a second copy of the L_i valid code words in L_{i+1}. So, the pathway from L_i to L_{i+1} is blocked and no further progress can be made until some other (non-blocking) codeword is decoded in L_i.

The suffix blocking condition occurs when decoding in a level L_i if the decoded codeword CW is a suffix of the prefix for L_{i+1} and the rest of the decoded sequence in lower level matches the prefix for L_{i+1}. Not all suffixes of a next level prefix cause blockage. For example, suppose the decoder is decoding in level L_i and the prefix for L_{i+1} is 101. If the decoded bit stream before reaching L_i can be …1 and the CW decoded in L_i is 01 (a suffix of 101) then this is a suffix blocking condition because there is a match with the prefix for L_{i+1}. Otherwise, if there is no match, the suffix condition is non-blocking and does not contribute anything to the synchronization delay. The prefix and suffix blocking conditions are described further below with reference to the code set C_{02} as shown in Table 1.

[1] T-codes are families of self-synchronizing VLC that have been used to model the self-synchronizing performance of many VLC reported in the literature through such simple operations as leaf node expansion / reduction [2].

Table 1 Part of the code construction / decoding table for the code set C_{02}

L_0		L_1		L_2		L_3		
		Prefix 0		Prefix 1		Prefix 00		
0	→	-		-		-		
1		1	→	-		-		
		00		00	→	-		
		01		01		01		Suffix of L_4 prefix
				-		-		
				11		11		
				100		100		
				101		101	→	L_4 prefix
						-		
						-		
						0000		
						0001		
						-		
						0011		
						00100		
						00101		

It is important to note that although one might be tempted to avoid suffix conditions as much as possible during code construction by always selecting current level prefixes from the first few of the previous level code words, this would not work in practice for two reasons. First, this approach is overly simplistic as it would rule out many code sets that could be used for accurate source matching (i.e. matching of average codeword length $\bar{L}$ or subgroup with source entropy H(S)). Second, simulation results have suggested that some high augmentation code sets having strong self-synchronization performance have suffix conditions occurring (especially in lower levels). The converse has also been found to be true (Higgie, 1996). So, the search for the "best" synchronizing codes entails the analysis of many more code sets than a restricted search space that one would like.

When using Table 1 for code construction, one observes that in the transition from L_0 to L_1, the L_1 prefix 0 is sacrificed in L_1, and is used to precede a second copy of all the code words in L_0 including the prefix itself. Then, from L_1 to L_2, the L_2 prefix is sacrificed, and is used to precede a second copy of all the code words in L_1, including the prefix itself. The process continues on to L_4 (not shown), when a total of seventeen code words are created.

For simple T-codes, the number of valid code words is $2^n + 1$, where n is the augmentation level. Incidentally, the augmentation level 3 $S_{0,1,00}^{1,1,1}$ T-code is obtainable from Table 1 as the nine code words under the L_3 column, i.e. $S_{0,1,00}^{1,1,1} = \{01, 11, 100, 101, 0000, 0001, 0011, 00100, 00101\}$. Similarly, lower augmentation level T-codes, such as $S_{0,1}^{1,1}$ can also be simply read off from the appropriate columns of the table.

Decoding follows the same trails as the augmentation for code construction. When a bit error occurs and synchronization is disrupted, the decoder receives a bit sequence of ones and zeros that in general do not appear as any valid codeword. Referring to Table 1 again, it is clear that if the decoder only receives zeros in the bit stream, decoding is blocked in L_0. In fact, no progress can be made until a one arrives at the decoder. This brings about a transition from L_0 to L_1. Decoding is blocked again at L_1 because the L_2 prefix is one. Further progress can only be made if the decoder receives either the bit sequence 00 or the bit sequence 01. This goes on until the final augmentation level is reached (in this case L_4). At this point a final level codeword boundary location is found and resynchronization occurs.

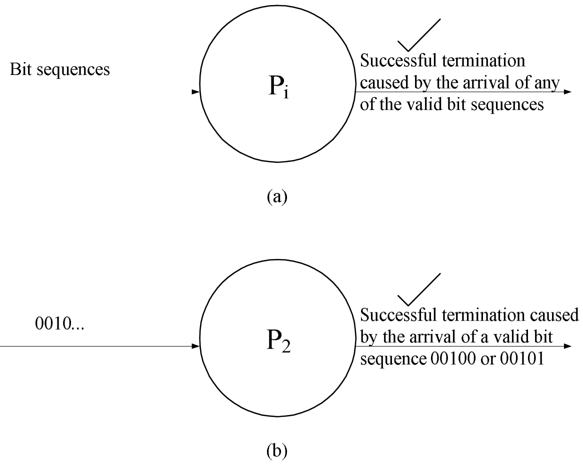
Clearly, by associating each decoding level to a process (i.e. representing decoding in level L_i by the process P_i), one can describe the whole decoding process from L_0 to L_n for an augmentation n code set using the sequential composition P_0 ; P_1 ; …; P_n. Further, for any process P_i, $\forall$ i : 0 … n, the refusal set contains all prefix or suffix blocking bit sequences in the corresponding L_i. For example, the refusal set of P_3 that corresponds to decoding in L_3 in Table 1 contains the elements 01 and 101, both of which cause blockage in that level of decoding.

In other words, ∀ i: 0 … n - 1, whenever a process P_i terminates and passes control to P_{i+1}, it means decoding has successfully progressed from level L_i to L_{i+1}. Since all VLC considered in this study are statistically synchronizable, successful termination of each P_i, ∀ i: 0 … n - 1and transfer of control to P_{i+1} is guaranteed to occur in finite time.

The process P_i, which is illustrated in Figure 2(a), therefore represents a generalization of the sequentially composed processes. This generalized process P_i acts like a filter, which inhibits blocking bit sequences (next level prefix and current level suffixes) while allowing valid bit sequences to pass through. The latter is effected by a successful termination of P_i and passing of control to the next subsequent process P_{i+1} until the final level is reached, which amounts to a successful decode of a valid codeword.

Figure 2(b) uses the process P_2, which represents decoding in level 2, to illustrate this point. Suppose the first four bits received are 0010. Then, the next bit that arrives (regardless of whether it is a 0 or 1) will effect a successful termination of P_2 and control is then passed to P_3. The set of refusals of P_2, now denoted by X_2 for convenience, contains the invalid (blocking) bit sequences. Thus, 00 and 1 are all members of X_2.

Figure 2 Generalization of sequentially composed processes


Bit sequences
P_i
Successful termination caused by the arrival of any of the valid bit sequences
(a)
0010...
P_2
Successful termination caused by the arrival of a valid bit sequence 00100 or 00101
(b)

It is through this generalization that ensures the scalability of this approach. For example, analysis and visualization of an augmentation 8 T-code set would be quite difficult (if not near-impossible) using the traditional tree approach as described in (Fong and Higgie, 2002). Using the stable failure approach, the whole decoding process is simply described by P_0; … ; P_8, where each P_i ; ∀ i : 0 … 8 is characterized by the corresponding (tr ,X) pair.[2]

As a further example, consider the code set C_{03}, which is the augmentation 10 simple T-code set $S^{1,1,1,1,1,1,1,1,1,1}_{1,10,111,111110,111110110,1111100,111111,1110,11111011,0}$. This code set has 1025 code words. Such a large code set is generally more than adequate for many multimedia applications. Interestingly, the shortest codeword is only 2 bits, whereas the longest is 47 bits. This kind of codeword length distribution can be very useful when the probabilities of occurrence of the various source symbols tend to be quite extreme: from very high to very low.

This code set distinguishes itself from the above not only just in terms of the number of code words, but also in the very different selection of prefixes. This is a deliberate effort to ensure generality of the examples presented, and applicability of the proposed method. Using the stable

[2] Information on the (tr , X)$_i$ pairs is required only for P_i , ∀ i : 0 … n - 1, which means 0 … 7 in this case. The last process P_n in the sequential composition (P_8 in this case) means decoding in the final level and signals the reestablishment of synchronization. This process therefore corresponds to the S state in the corresponding decoding tree.

failure approach, the whole decoding process is simply described by P_0; … ; P_{10}. The refusal sets for the code set C_{03} are listed in Table 2.

The above examples illustrate how the CSP stable failure model can be used to concisely describe the decoding / resynchronization process of large code sets. In fact, the example code set C_{03} is significantly larger than those typically found in the literature. For example, it would be near impossible to analyze such large code sets using a traditional tree algorithm as described in (Fong and Higgie, 2002). In the past, the decoding / resynchronization behaviors of such large code sets were typically studied using simulation techniques.

Evidently, it is quite possible (at least conceptually) to extend this method further to cover even larger code sets, including those that are far too large for general applications. For example, code sets as large as augmentation degree 16 are more for theoretical analysis than practical applications. Yet, the whole decoding process can be simply described by P_0; P_1;…; P_{15}; P_{16}, though it would of course take a little effort to collect all the $(tr, X)_i$ information. However, this would still be far more feasible — and elegant — than even trying to describe the decoding process of such a large code set using a traditional tree approach.

Table 2 Refusal sets for the code set C_{03}

level i	C_{03} refusal sets
0	1
1	10, 0
2	111, 11
3	111110, 1110, 110, 0
4	111110110, 110, 0
5	1111100, 111100, 100, 0
6	111111, 11111, 11
7	1110, 110, 0
8	11111011,11
9	0

In summary, the approach presented in this subsection, which applies the stable failure model to a sequential composition of processes with events made up of valid bit sequences, provides an elegant way to reason about the VLC decoding / resynchronization process. It is also highly scalable through the generalization of each process Pi that makes up the composite process. This composite process describes the complete decoding / resynchronization process for any VLC that possess the properties supposed in this dissertation (i.e. exhaustive, instantaneously decodable, statistically synchronizable, and so on). Further, the final trace of the composite process leads to the identification of synchronizing sequences (described in the next subsection), and can give a quantitative measure of the synchronization performance of the code set in question.

4 Description of synchronizing sequences

A byproduct of the above discussion is that the traces associated with the processes can be used to describe the so-called synchronizing sequences. It has been known for a long time (Rudner, 1971) that some bit patterns can bring about rapid recovery of synchronization in the course of bitstream decoding following a lock loss. These bit patterns occur frequently and naturally in messages encoded with a well-designed self-synchronizing VLC. Unlike insertions of resynchronization markers, synchronizing sequences that appear as a result of well-constructed codes do not add any overhead to the encoded bitstream. This could be an important consideration in low-bandwidth applications such as internet videoconferencing or mobile communications.

Normally, we would like synchronizing sequences to be among the shortest bit patterns that bring about rapid resynchronization following a lock loss. Clearly, from a practical perspective, long bit sequences that can also bring about resynchronization are just not so useful.

We can describe synchronizing sequences by following the Model B way of thinking (using codewords as events) within the Stable Failure framework as described in Section 3. For example, using code set C_{02} to illustrate, we may observe traces such as $\langle 1\rangle$, $\langle 0, 1\rangle$, $\langle 0, 0, 1\rangle$ before the process P_0 terminates successfully and control passed to P_1.

Now, suppose we again want short synchronizing sequences as usual. This means we may simply want to store the shortest trace up to this point, i.e. $\langle 1\rangle$. Likewise, we may observe traces such as $\langle 00\rangle$, $\langle 1, 00\rangle$, $\langle 1, 01\rangle$, $\langle 1, 1, 00\rangle$ before P_1 terminates, but we are mostly interested in the shortest traces, i.e. $\langle 00\rangle$ and $\langle 01\rangle$. So, we continue this procedure for all other processes P_i that make up the sequential composition. Synchronizing sequences are then obtainable by concatenating these stored traces, especially (but not necessarily) the shortest ones. For example, a synchronizing sequence may be formed by concatenating the traces $\langle 1\rangle$ (from P_0), $\langle 00\rangle$ (from P_1), $\langle 01\rangle$ (from P_2) and $\langle 01\rangle$ (from P_3). This concatenation results in the trace $\langle 1, 00, 01, 01\rangle$, from which we can deduce that 1000101 is a synchronizing sequence. This bit sequence will certainly bring about rapid resynchronization when presented to the decoder. However, this may not be necessarily among the shortest possible. This is a point worth exploring further.

In fact, we can observe from the above example that we need to perform additional checking to remove multiple instances of the same trace from consecutive processes in the sequential composition. This corresponds to taking into account the so-called straight through situations during normal decoding, which has the effect of shortening the bit sequence required to resynchronize because the same subsequence can be used to progress from one decoding level to the next subsequent level.

Indeed, in the present context, if we eliminate duplicate traces from consecutive processes we do end up with shorter synchronizing sequences. Referring back to the above example, we find that we have captured the same trace $\langle 01\rangle$ for the consecutive processes P_2 and P_3. So, we can eliminate the second occurrence of $\langle 01\rangle$ in the final concatenation to yield the trace $\langle 1, 00, 01\rangle$, from which we can deduce that 10001 is also a synchronizing sequence. Clearly, this shortest bit sequence can bring about even more rapid resynchronization that the previous sequence, although both can be described as synchronizing sequences. A careful inspection reveals that 10001 is among the shortest synchronizing sequences possible for this particular code set.

From the above discussion, we can see that CSP traces can be applied to describe synchronizing sequences elegantly by observing the resultant traces from the stable failure model as described in Section 3. This means concatenating the tr part of the $\{tr, X\}_i$ pair associated with each P_i that makes up the sequential composition describing the whole decoding mechanism.

Again taking the code set C_{02} as an example, the shortest sequences that can bring about valid decoding in L_4 are 10001, 10011, 100100 and 100101. These bit patterns are synchronizing sequences because a codeword boundary location is found and resynchronization occurs upon decoding the last bit of any of these sequences, just as though the last bit of an inserted resynchronization marker had been decoded. Importantly, these synchronizing sequences are obtainable without the need for a traditional tree-based method, which is quite clumsy in comparison.

Evidently, the augmentation algorithm propagates codeword boundary information from L_i to L_{i+1}, and so on. In particular, every L_i codeword (except the L_{i+1} prefix) becomes a valid L_{i+1} codeword or a suffix of it. It is clear that this augmentation propagates codeword boundary information.

In fact, by analyzing the resultant traces from the stable failure model, it is possible to find, for any augmentation level n, T-code sets that have more desirable distributions of synchronizing sequences than others. For example, the resultant traces tend to be shorter if the L_i prefix is chosen such that its position is 2^i larger than the L_i-1 prefix, for as many of these prefixes as possible subject to constraints such as matching average codeword length $\bar{L}$ with source entropy H(S) (i.e. matching of subgroups with H(S)). For example, the T-code set $S^{1,1,1,1}_{0,1,00,0000}$ (positional difference of L_2 and L_3 prefixes is 8) performs better than $S^{1,1,1,1}_{0,1,00,101}$ (positional difference of L_2 and L_3 prefixes is 5) in terms of minimizing the average synchronization delay. This observation agrees with earlier findings reported in (Fong and Higgie, 2000).

5 Conclusion

Variable length codes (VLC) have found widespread applications in practical situations, ranging from data transmission to compact storage of data. Loss of synchronism remains an important issue, even when error resilience tools such as insertion of synchronization markers are used. CSP, which is a well-established process algebra used by software engineers to reason about complex systems, has been found useful in modeling the synchronization of VLC.

After a brief presentation of the fundamentals of CSP, we explored the applicability of the CSP stable failure model to the blocking conditions encountered by the decoder during the decoding / resynchronization process of VLC. The basic idea was to use the set of refusals to model the blocking events during any stage of decoding. The first part of the discussion established that the CSP stable failure model was not applicable to the "atomic" way of describing the events because the set of refusals would always be empty for the exhaustive codes under consideration.

On the other hand, the CSP stable failure model was shown to be very much applicable to situations where events are made up of valid code words, or more generally bit sequences. In particular, by considering each level of decoding as a CSP process with an associated trace tr and refusal set X (i.e. the pair (tr, X)), the whole decoding process from the point of lock loss (decoding level zero represented by process P_0) to the point of resynchronization (decoding in highest augmentation level n represented by process P_n) can be conveniently modeled as sequential composition of processes P_0; P_1 … ; P_n. Moreover, the resultant traces could be used to obtain synchronizing sequences and the refusals describe the various blocking conditions encountered throughout the whole decoding process. This provides an elegant way of describing, in principle, the decoding mechanisms of arbitrarily large code sets. Examples using practically useful code sets (with up to 1025 code words as in the case of the code set C_{03}) — and the fact that the prefixes and expansion parameters were chosen quite randomly — validate this point.

We have also established a link between the stable failure model and synchronizing sequences. In particular, Section 4 has shown that the resultant traces from the stable failure model in Section 3 can be used to precisely describe the synchronizing sequences.

If resynchronization markers that are "artificially" inserted into the bit stream to "force" the decoder to resynchronize, then synchronizing sequences exist "naturally" in the encoded bit stream as concatenation of valid code words as a matter of course during normal encoding with a well-designed VLC. It then stands to reason that a VLC that gives rise to frequent natural occurrences of synchronizing sequences is also one that can be said to exhibit strong synchronization performance.

The examples with randomly selected parameters have been used to illustrate the various ideas throughout the paper to validate general applicability. Future research could extend this work by introducing observability (through hiding) and probability distributions to describe more error scenarios.

References

[1] Capocelli R. M., Santas A. A. D., Gargano L., and Vaccaro U.. On the construction of statistically synchronizable codes, *IEEE Trans. Inf. Theory*, 38(3):407-414, 1992.

[2] Ferguson T. J. and Rabinowitz J. H.. Self synchronizing hu_man codes. IEEE Trans. Inf. Theory, 30(4):817-825, 1984.

[3] Fong A. C. M. and Quay C.. Application of self-synchronizing codes to FLEXTM suite message encoding. Motorola Tech. Dev., 40:68-72, 2000.

[4] Fong A. C. M. and Higgie G. R.. Identification of T-codes with minimal average synchronization delay. IEE Proc. Computers and Digital Techniques, 147(4):237-241, 2000.

[5] Fong A. C. M., Higgie G. R., and Fong B.. Multimedia applications of self-synchronizing codes. In Proc. IEEE Int. Conf. on IT: Coding and Computing, pages 519-523, Las Vegas, NV, USA, 2001.

[6] Fong A. C. M. and Higgie G. R.. Using a tree algorithm to determine the average synchronization delay of self-synchronizing T-codes. IEE Proc. Comput. Digit. Tech., 149(3):79-81,2002.

[7] Fong A. C. M., Higgie G. R., Fong B., and Hong G. Y.. Analysis of the decoding process of T-codes. IEE Proc. Commun., 149(4):202-206, 2002.

[8] Fong A. C. M. and Simpson A.. Using CSP to model the synchronization process of variable length codes. IEE Proc. Commun., 153(2):195-200, 2006.

[9] Higgie G. R.. Database of best T-codes. IEE Proc- Comput. Digit. Tech.,, 143:213-218, 1996.

[10] Hoare C. A. R.. Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.

[11] Huffman D. A.. A method for the construction of minimum-redundancy codes. Proc. IRE, 40:1098-1101, 1952.

[12] Roscoe A. W.. The theory and practice of concurrency. Prentice Hall, 1997.

[13] Rudner B.. Construction of minimum redundancy codes with optimum synchronization property. IEEE Trans. Inf. Theory, 17:478-487, 1971.

[14] Takishima Y., Wada M., and Murakami H.. Error states and synchronization recovery for variable length codes. IEEE Trans. Commun.,42(2/3/4):783-792, 1994.

[15] Titchener M.R.. The synchronization of variable-length codes. IEEE Trans. Inf. Theory, 43(2):683-691, 1997.

[16] Zhou G. and Zhang Z.. Synchronization recovery of variable-length codes. IEEE Trans. Inf. Theory, 48(1):219-227, 2002.