FORMAL MODELLING OF CHANNEL-AND SOURCE-CODING ALGORITHMS FROM CODING THEORY

Thomas Michlmayr

October 20, 2026

Problem Domain

- Today's digital world consists of transmitting and sending data e.g.:
  - phone calls
  - internet connection
  - messages
  - ...
- For such digital data it is of interest to correct errors, if ones should occur, and to reduce the size of data.
- How do we achieve such comfort:
  1. Find proper algorithms for coding and decoding/transmitting messages. (Information Theory)
  2. Prove their correctness mathematically. (Mathematics)

Goals of the Thesis

- The goal of the thesis is to
  - mathematically specify and
  - implement

  selected concepts and techniques from information theory in the language of the mathematical model checker **RISCAL(RISC Algorithm Language)**
- The thesis covers 7 algorithms and explicitly shows 41 RISCAL checks of theorems and postconditions.

Information Theory

- A communication system is defined as the path which data travels from a source to a destination (Source $\rightarrow$ Transmitter $\rightarrow$ Channel $\rightarrow$ Receiver $\rightarrow$ Destination).
- In the real world, sometimes data gets corrupted (noise). This noise appears (typically) in the channel.
- To determine how much information can be sent via a channel, following size is defined:

Definition

Let $C = \lim_{T \to \infty} \frac{\log(N(T))}{T}$ where $N(T)$ is the number of signals of duration T and $\log(N(T))$ is the number of bits needed to identify each possible signal. Then C is called **(channel) capacity**. The capacity is measured in bits per time unit.
If all symbols have equal duration, the capacity can be interpreted as bits per symbol.

- To determine how much information is contained in a message, how uncertain an event is, and how many possible outcomes exist, following size is defined:

Definition

Let $H(p_1, ..., p_n) = -\sum_{i=1}^{n} p_i \log(p_i)$. We call H **Entropy** and sometimes write H for $H(p_1, ..., p_n)$ and $H(x)$ for the entropy of random variable x. Entropy is measured in bits.
When expanding the theroy of a single symbol to a sequence G_N of symbols of length N, the entropy is defined as $H = \lim_{N\to\infty} \frac{G_N}{N}$ and is also called **entropy rate** and measured in bits per symbol.

Information Theory

- To prevent total loss of data, different approaches can be made:
  - Increase channel stability against external influences (not part of my thesis)
  - Coding data such that errors can be detected and corrected (**E**rror **C**orrecting **C**odes) $\equiv$ **Channel Coding**
    - Deals with Channel capacity.
    - Is basically about adding redundancy.
- Reducing data size can be understood as:
  - Coding data such that it gets transmitted more efficiently (Compression) $\equiv$ **Source Coding**
    - Deals with Entropy.
    - Is basically about removing redundancy.

Mathematics

- For channel coding, our system can be mathematically understood as functions such that:
  $\forall x, y : somewhatcorrupted(y, encode(x)) => decode(y) = x$ with x as data
- For source coding, our system can be mathematically understood as functions such that:
  $\forall x : decode(encode(x)) = x$ with x as data
- Thus, checking the algorithms validity, first order logic (FOL) properties have to be proven.
- FOL on infinite domains is undecidable $\Rightarrow$ there is no algorithm which can determine the logical correctness of the algorithms for every input, hence this way of proving is not automatable.
  $\Rightarrow$ restriction to finite domains (models), **Formal Model Checking**
- Formal Model Checking is a technique for verifying programs operating on finite domains by checking all reachable states with respect to given specific properties.

Mathematics - RISCAL

- RISC Algorithm Language (**RISCAL**) is a suitable tool for Formal Model Checking.
- Provides language where algorithms and logical properties can be described with logical formulas in FOL.
- Allows to automatical check the validity of models of fixed size via going through the whole state space.
- Even provides counterexamples when algorithms are not valid.
- Enables creation of own data types, predicates (logical statements), functions/procedures which satisfy predicates, and theorems.

Channel Coding

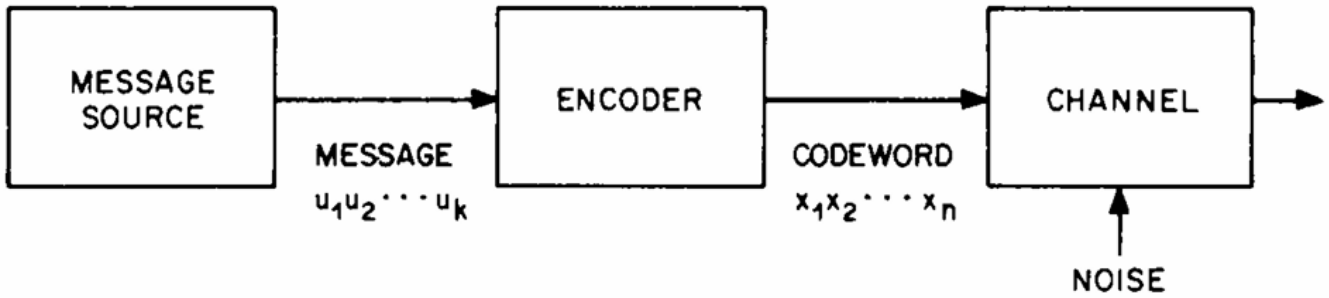


Figure 1: Transmitting and encoding messages

- A message u which is built by k symbols $u = u_1, u_2, ..., u_k$.
- The encoded message has n symbols, known as **length**.
- The encoder appends $n - k$ **parity bits** such that we receive a codeword $x = x_1, x_2, ..., x_n$ after encoding our message.
- The encoded message consists of the original k message symbols and the $n - k$ parity bits.
- k is also called **dimension**.

- Core Principle for parity bits:

$$H \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} = 0$$

- H denotes the $(n-k) \times n$ **parity check matrix** which is defined as $H = [A|I_{n-k}]$, where A is a $(n-k) \times k$ matrix and I_{n-k} is the unity $(n-k) \times (n-k)$ matrix.
- The components of A depend on the expectations of the algorithm.
- To achieve error detection, the additional parity bits have to be set correctly, i.e., with given A the equation must be solved for the unknown parity bit variables. Hence, these equations for each component are called **parity check equations**

- The parity check equations provide a complete characterization of valid codewords. We therefore define the code as the set of all vectors satisfying these equations.

Definition

Let H be a binary parity-check matrix. The corresponding **code** C is defined as $C = \{x \in \mathbb{F}_2^n : Hx^T = 0\}$.
Furthermore, a code is called **linear**, if for all codewords $x, y \in C$, the sum $x + y$ is also a codeword in C, and for every field element c, cx is also a codeword in C.
For these x, y, and c it holds $H(x+y)^T = Hx^T + Hy^T = 0$ and $H(cx)^T = cHx^T = 0$.

Definition

Let C be a linear code with length n and dimension k. Then C is called a $[n, k]-$ code with **rate** $R = \frac{k}{n}$. The code consists of 2^k codewords.

We now consider the error vector $e = y - x = e_1 e_2 ... e_n$, where y is the received vector and x the codeword.

Definition

The number of positions with different values between two vectors x and y is called **Hamming distance** denoted by $dist(x, y)$.

- Single messages/vectors can be also characterized by the number of ones:

Definition

The number of nonzero positions of a vector x is called **Hamming weight** denoted by $wt(x)$.

- Considering the previous definitions, it follows that for the error vector $x - y$ between the vectors x and y, it holds that $dist(x, y) = wt(x - y)$.

- While the Hamming distance compares two individual codewords, a code C should also have a characteristic distance. Therefore, the minimum distance of a code is defined as the smallest Hamming distance between any two distinct codewords:

Definition

Let C be a linear code. The **minimum distance** of C is defined as $mindist(C) := min_{x,y \in C, x \neq y} dist(x, y) = min_{x,y \in C, x \neq y} wt(x - y)$.

Definition

Let C be a linear code with length n, dimension k and $mindist(C) = d$. Then C is called a $[n, k, d]-$ code.

- Thus, every pair of valid codewords differs in at least d positions. Intuitively, a larger minimum distance results in easier distinction between two codewords. The minimum distance determines important properties of a code:

Theorem

The minimum distance of a linear code is the minimum weight of any nonzero codeword.

- Hence, determining the minimum distance does not require comparing every pair of codewords. Instead, it is sufficient to inspect the weights of all nonzero codewords.

- Finally, the minimum distance determines the error-correcting capability of a code:

Theorem

*A linear code C with $\text{mindist}(C) = d$ can correct $\lfloor \frac{d-1}{2} \rfloor$ errors. Furthermore, if d is even, the code can **simultaneously** detect $\frac{d}{2}$ errors and correct $\frac{d-2}{2}$ errors.*

Theorem

*A linear code C can **simultaneously** correct at most a errors and detect up to b errors with $b > a$ if and only if $\text{mindist}(C) \geq a + b + 1$. In particular, every code with minimum distance d can detect up to $d - 1$ errors.*

Channel Coding - Even Weight Code

- We are now considering the **even weight code**.
- Here, the codeword with length n must contain an even number of ones by adding just one parity bit. Hence, $n - k = 1$ holds.
- Therefore, H is just a row vector as its components only represent a single equation.
- For even weight, H consists of components with value 1, since all bits have to be added to check for an even number of ones.
- An even weight code C has $mindist(C) = 2$. Even weight codes are $[n, n-1, 2]$ codes.
- Even weight codes can only detect single errors.
- We can now implement this code in RISCAL:

Channel Coding - Even Weight Code RISCAL

The goal of ensuring the functionality of the established theory, consists of showing first that encoded messages can be fully decoded. Second, the capability of detecting single errors must be checked. This can be implemented via theorems (*DecodeEncode* and *DetectSingleBitError*) in RISCAL which will be tested in the end of the RISCAL code.

```
theorem DecodeEncode(m: Message) ⇔ decode(encode(m)) = m;
theorem DetectSingleBitError(m: Message, c: Codeword) ⇔
singleError(encode(m),c) ⇒~validCodeword(c);
```

To reach this goal various implementations have to be done.

Channel Coding - Even Weight Code RISCAL

```
val K: Nat;
val N = K+1;
type Bit = Nat[1];
type Message = Array[K,Bit];
type Codeword = Array[N,Bit];
```

As model for the theory data types need to be introduced. We define the types *Bit*, *Message* and *Codeword* where *K* indicates the bit size the original message has and *N* the according codeword size. *K* is a RISCAL model parameter which is set in the RISCAL environment.

Channel Coding - Even Weight Code RISCAL

```
proc encode(m: Message): Codeword
ensures validCodeword(result);
{
var c: Codeword := Array[N,Bit](0);
for var i:Nat[K] := 0; i < K; i := i+1 do
c[i] := m[i];
c[N-1] := parityMessage(m,K-1);
return c;
}
```

The procedure *encode* shows the simple encoding process for even weight codes. For input messages a separate codeword array is created and filled with the information bits. The parity bit is its last element and is calculated by *parityMessage*. Encode ensures, that the process produces a valid codeword via the predicate *validCodeword*.

Channel Coding - Even Weight Code RISCAL

```
pred validCodeword(c: Codeword) ⇔ parityCodeword(c,N-1) = 0;
```

validCodeword is the property that valid codewords for Even Weight Codes have even parity.

Channel Coding - Even Weight Code RISCAL

```
fun xor(a: Bit, b: Bit): Bit =
if a = b then 0 else 1;

fun parityMessage(m: Message, i: Nat[K-1]): Bit =
if i = 0 then m[0] else
xor(parityMessage(m,i-1), m[i]);

fun parityCodeword(c: Codeword, i: Nat[N-1]): Bit =
if i = 0 then c[0] else
xor(parityCodeword(c,i-1), c[i]);
```

By logical XOR operations, the parity of bit sequences can be evaluated. Therefore, a *xor* function has been established. For checking the parity of messages and codewords, separate recursive functions have been implemented.

Channel Coding - Even Weight Code RISCAL

```
proc decode(c: Codeword): Message
requires validCodeword(c);
{
var m: Message := Array[K,Bit](0);
for var i:Nat[K] := 0; i < K; i := i+1 do
m[i] := c[i];
return m;
}
```

decode decodes the codeword by extracting the bits coming before the parity bit. Here, *validCodeword* is required for input codewords, which is ensured by the encoding procedure.

Channel Coding - Even Weight Code RISCAL

```
pred singleError(c1: Codeword, c2: Codeword) ⇔
∃i:Nat[N-1].
c1[i] ≠c2[i] ∧
(∀j:Nat[N-1] with j ≠i. c1[j] = c2[j]);
```

singleError indicates the property if just one single error occurred during data transmission.

Channel Coding - Even Weight Code RISCAL

```
Executing DecodeEncode(Array[ℤ]) with all 16777216 inputs.
Execution completed for ALL inputs (327631 ms,
16777216 checked, 0 inadmissible).

Executing DetectSingleBitError(Array[ℤ],Array[ℤ])
with all 33554432 inputs.
Execution completed for ALL inputs (309154 ms,
33554432 checked, 0 inadmissible).
```

For the RISCAL check of *DecodeEncode K=24* has been used. For *DetectSingleBitError* uses *K=12* to reduce execution time.

Source Coding - Run Length Encoding

- **Source Coding** is about reducing size of data.
- Data can be text files, bit streams, images, ...

Definition

Single elements of input data are called **data items** or **data symbols**.

Definition

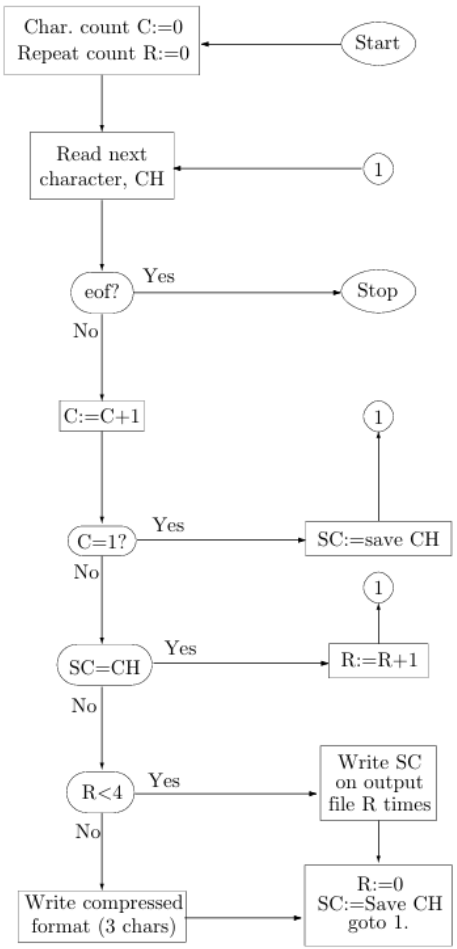
The number n of consecutive appearances of data items is denoted as the **run length**, **repetition count**, or **run**.

- We consider **Run Length Encoding (RLE)**.
- Here, sequences of the data are checked for consecutive appearances of data items, e.g., the text input *'a bbb c'* has data items *'a'* with runlength 1, *'b'* with 3 and *'c'* with 1.

Source Coding - Run Length Encoding

- The original sequence can now be written as: *'a @3b c'*: segments of consecutive items in the sequence are represented by an **escape character** (in the example the symbol *'@'*) followed by the run length and the data item itself.
- This method yields problems:
  - If run lengths are too short, the compressed data could be equally big as before or even bigger. Therefore, runs should only be encoded if their lengths are large enough to compensate for the additional encoding overhead.
  - The escape character may itself occur in the input data.

Source Coding - Run Length Encoding



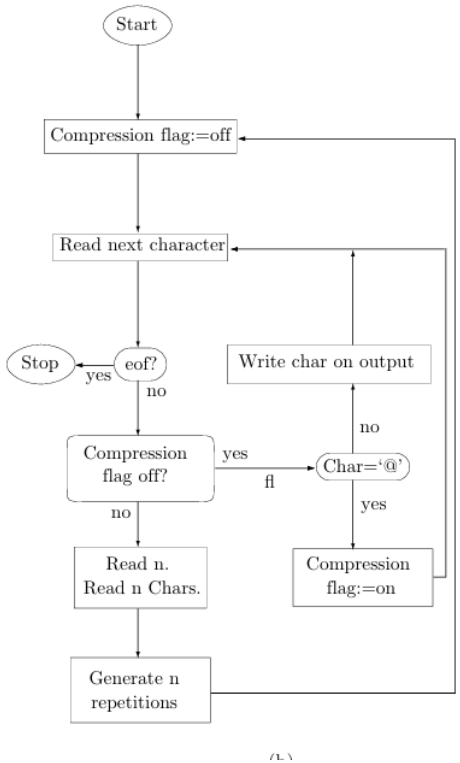


Figure 2: RLE encoding and decoding flow charts

Source Coding - Run Length Encoding

We assume that the encoder writes **RLE-tokens** to the output. Each token contains a compression flag, the run length, and the corresponding symbol. Symbols with run lengths of at least 3 are compressed.
The key theorem for source coding algorithms which must be satisfied is:

```
theorem decodeEncode(t:Text) ⇔ decode(encode(t)) = t;
```

Source Coding - Run Length Encoding

```
val N:Nat;
val A:Nat;

type Symbol = Nat[A];
type Index = Nat[N-1];
type Text = Array[N,Symbol];
type Count = Nat[N];
type ElementCount = Nat[N];
type Element = Record [compressed: Bool, count: Count,
       symbol: Symbol];
type EncodedText = Record [length: ElementCount,
             elements: Array[N,Element]];
```

First, the main types are established. The type *Element* represents one RLE-token. Its field *compressed* indicates whether the token represents a compressed run, *count* stores the number of repetitions, and *symbol* stores the corresponding symbol. The type *EncodedText* then represents the complete encoded text. Its field *length* specifies how many elements are actually used, while *elements* stores the corresponding RLE-tokens.

Source Coding - Run Length Encoding

```
pred run(t:Text,start:Index,count:Count,s:Symbol) ⇔
 start + count <= N ∧ ∀j:Count with j < count.
 t[start + j] = s;

pred completeRun(t:Text,start:Index,count:Count,s:Symbol)
⇔
 run(t,start,count,s) ∧ (start = 0 ∨ t[start - 1] ≠s) ∧
 (start + count = N ∨ t[start + count] ≠s);

fun runLength(t:Text,start:Index,s:Symbol):Count
requires runStartsAt(t,start,s);
ensures completeRun(t,start,result,s);
= choose c:Count with completeRun(t,start,c,s);
```

run specifies a sequence of consecutive occurrences ot the same symbol in text.
completeRun extends *run* by ensuring that the sequence cannot be extended to the left or to the right with the same symbol.
runLength determines the length of the complete run starting at a given position.

Source Coding - Run Length Encoding

```
proc encodeOneRun(t:Text,start:Index,
       count:Count):EncodedText
requires start < N ∧ count >= 1 ∧ start + count <= N ∧
completeRun(t, start, count, t[start]);
{
var result:EncodedText;

if count = 1 then result := encodeSingleRun(t,start);
else if count = 2 then result := encodeDoubleRun(t,start);
else result := encodeCompressedRunText(count,t[start]);

return result;
}
```

encodeOneRun selects the appropriate representation based on the run length. Runs of length one or two are stored as individual uncompressed elements, while runs of length at least three are represented by a single compressed element.

Source Coding - Run Length Encoding

```
proc encode(t:Text):EncodedText
requires N > 0;
ensures encodes(t,result);
{
var result:EncodedText = encodeFirstRun(t,0);
var count:Count = runLength(t,0,t[0]);

if count < N then result := encodeFrom(t,count,result);

return result;
}
```

encode starts the encoding process by encoding the first run of the input text. If the first run does not cover the entire text, the procedure *encodeFrom* continues encoding the remaining part. The postcondition *encodes(t,result)* specifies that the resulting encoded text is a valid encoding of the input text.

Source Coding - Run Length Encoding

```
proc decode(e:EncodedText):Text
requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
decodedLength(e) = N;
ensures ∀p:Index. decodeAt(e,p) = result[p];
{
var result:Text = Array[N,Symbol](0);
for var p:Position := 0; p < N; p := p + 1 do
result[toIndex(p)] := decodeAt(e,toIndex(p));
return result;
}
```

decode reconstructs the original text by calling *decodeAt* for every position in the decoded text. Each returned symbol is stored at the corresponding position in the result array.

Source Coding - Run Length Encoding

```
proc decodeAt(e: EncodedText,p: Nat[N]): Symbol
requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
p < decodedLength(e);
ensures decodedAt(e,p,result);
{
var result: Symbol;
choose result: Symbol with decodedAt(e,p,result);
return result;
}
```

decodeAt returns the symbol at position p in the decoded text. It uses *choose* to select a symbol satisfying the predicate *decodedAt*. This predicate states that there exists an encoded element containing position p whose stored symbol is the symbol returned by *decodeAt*.

Source Coding - Run Length Encoding

```
Executing decodeEncode(Array[ℤ]) with all 16384 inputs.
Execution completed for ALL inputs (123919 ms,
16384 checked, 0 inadmissible)
```

RISCAL checks for all possible texts this property for $N=7$ and $A=3$.

Further implemented Algorithms

- Channel Coding:
  - Hamming Code
  - Construction of new codes with old ones (Extending)
- Source Coding:
  - Golomb Code
  - Shannon-Fano/Huffman Code
  - Lempel-Ziv LZ77/LZ1

Summary

- RISCAL is a suitable tool for formally modelling and verifying coding algorithms
- The exhaustively checking of all states of the state space easily detects weak specifications or errors in the implementation.
- Formal modelling requires careful consideration of possible states and input combinations. Although the mathematical specifications of many algorithms are relatively straightforward, special cases and invalid inputs have to be explicitly addressed by predicates, preconditions, and other restrictions.
- The number of inputs for RISCAL executions grow rapidly with the size and number of parameters (longest execution took $1,5h$).
- The complete thesis is available at `https://doi.org/10.35011/risc.26-15`.