Explorations in Kahn Process Networks

Parsing, Processes, and More

Ian Daniher
Olin College of Engineering
ian.daniher@students.olin.edu

ABSTRACT

Kahn's 1974 work describes simple semantics for describing the behavior of collections of communicating processes. In this paper, the run length encoding algorithm is shown to be describable as Kahn process networks, and the implications of this are discussed. Code examples in the Rust programming language are included and analyzed.

1. INTRODUCTION

Computers can act as tools to enhance our ability to sense and understand the world around us. Unfortunately, many programming paradigms do not adequately support concurrency or stream operations, limiting their utility as tools to enhance human abilities. The well-known environments that do easily offer the ability to execute parallel stream manipulations, such as Simulink, GNUradio, and LabVIEW have decades of dogma and multiple layers of complex, underdocumented abstractions separating a user from the concise implementation of formal roots. While access to tools providing commodity hardware with more sophisticated eyes and ears has increased dramatically in recent years, the software tools to make use of these new senses have lagged behind. Kahn Process Networks are a simple and rigorous paradigm for describing and reasoning about the processing of information streams, research implementations of which have been used to carry out real-time concurrent stream manipulation[1].

While implementations of such tools leaves something to be desired, extensive amounts of work have been done in the field for many decades. Gilles Kahn was a French computer scientist who, like Scott and McCarthy, worked to formalize the nascent field of computer science. In his 1974 paper titled "The Semantics of a Simple Language for Parallel Processing," Kahn puts forth a model capable of describing the interactions of communicating parallel processes, with independent processes communicating exclusively via unidirectional channels. Kahn's channels act as first-in, first-out (FIFO) queues with unlimited buffers. A write to such a channel is non-blocking and completes immediately. A read from such a channel blocks and only returns a token when data is available. The programming language "Rust" is a C-like language self-described as "..safe, concurrent, practical..."[2]. It provides as primitives necessary building blocks to instantiate networks according to Kahn's description.

The Rust language is a new systems language sponsored by Mozilla, the organization responsible for the Firefox web-browser. It consists of a compiler, standard library, and runtime, all written in the language itself. The Rust runtime abstracts away the implementation of efficient task scheduling and communication between processes, which has been the focus on many other research projects on KPNs to date. [5] [7] [8] The standard library exposes the creation and manipulation of "streams" - unidirectional communication channels akin to Kahn's channels, and "tasks," isolated independent processes which are executed in parallel in either separate or shared operating system threads.

The first page of [6] features an extended syntax ALGOL program 'S' implementing the process network paradigm. A Rust implementation of 'S,' complete with functional schematic, can be found in Appendix A.

2. CONTEXT: PARSERS

The parsing logic discussed here and illustrated in Appendix B exists to extract information from samples of digitally encoded signals transmitted on either analog or digital media. Transmissions are typically captured at a rate many times the maximum symbol rate, without synchronization between the transmitter and receiver. As such, a parser needs to identify the start of a transmission, reduce a set of input samples by many times, and map encodings and modulations to their inverse. This needs to be carried out efficiently, as in many situations the ratio between the input data rate and the processor clock is less than 1000x, necessitating processor-efficient reduction of input.

Extensive work has been done on the parsing of digital bitstreams, commonly described using regular expressions and implemented using either finite state machines or backtracking variants. Common implementations of regular expressions [3] are designed to operate upon finite length string and can operate in time increasing exponentially with the size of input. While alternative implementations addressing these issues do exist[4], they are still constrained in terms of expected input, and are generally poorly suited to the de-

coding and demodulation tasks associated with RF parsing.

The parser logic in Appendix B functions with negligible connections to regular expression / DFA / backtracking parsers with roots in automata theory, but is capable of accepting an input stream and extracting the intended information, returning it as output. Initial implementations were created operating upon finite length inputs, parsing in time approximately proportional to the number of input elements. These initial implementations required refinement due to the desire for parser functions capable of operating upon infinite length input, such as a stream provided by a software defined radio. Run length encoding, discussed below, shares the same general form as the parsing logic of Listing 3, and is shown to be functionally (and possibly mathematically) equivalent with a valid KPN construct operating upon infinite length inputs, allowing us to make generalizations as to whether or not the instantiated system will terminate and on what domain the input and output are well defined.

3. RUN LENGTH ENCODING

Run length encoding is a compression function accepting an input sequence x and returning an output sequence of two-tuples y of the form (v, ct), each representing the value and length of successive homogeneous sub-sequences.

RLE provides a concise means to reason about sequences of sampled states by collapsing sequences of samples containing the same measured state to a single two-tuple containing state and duration. After an input has been discretized, it can be reduced by applying run length encoding, and the resulting sequence of two-tuples can be inspected.

When describing run length encoding as a KPN node, the denumerably infinite input sequence is assumed to be received successively one token at a time, and output sent upon observed line-state transition. The sequences described as the inputs and outputs of the run length encoding function are the line state histories of the input and output channel, respectively.

The behavior of this function on finite input sequences merits further investigation, but preliminarily, can be described as follows.

$$rle(x) = y \tag{1}$$
$$|x| \in \mathbb{N} \tag{2}$$
$$|y| < |x| \tag{3}$$
$$max(y[ct]) < |x| - 1 \tag{4}$$
$$|y| \in \{\mathbb{N} \cup 0\} \tag{5}$$

Application of the function on finite length inputs yields results ignoring the terminating run, responsible for the constraints described above.

The expected behavior of run length encoding for infinite sets is as follows

$$|x| \leftrightarrow |\mathbb{N}| \tag{6}$$
$$|rle(\mathbb{N})| \leftrightarrow |\mathbb{N}| \tag{7}$$
$$\forall a \in dom_x, x = (a, a, a....) \implies |y| = \emptyset \tag{8}$$
$$|x| < |x| \implies |y| < |x| \tag{9}$$

For the sake of unifying the semantics between a KPN expecting infinite input and a function applied to finite input, for all finite input sequences, let the terminal value be considered to be extended infinitely many times.

An implementation of run length encoding as a process in a KPN follows.

Listing 1: run-length encoding as a process

```
1  pub fn rle(U: Port<uint>, V: Chan<Run>){
2    let mut x: uint = U.recv();
3    let mut i: uint = 1;
4    loop {
5      let y = U.recv();
6      if (x != y) {
7        V.send(Run { v: x, ct: i });
8        i = 1;
9      }
10     else {
11       i = i + 1;
12     }
13     x = y;
14   }
15 }
```

This process has two internal states, one initialized to the first token received, one initialized to '1.' In the main loop, The value read from channel is compared against the stored value. If it is different, it transmits a token of the stored value and the stored count. If it is the same, the stored count is incremented. The value read from the channel is then stored.

This process is intended to operate as a part of a process network and used in conjunction with parsers of the form of Appendix B, Listing 3. As such, the process ought to be aligned with the expected behavior for a valid entity in a KPN.

In section 2.2 of [6], Kahn describes the semantics of valid processes in his network, namely, that they must act as continuous mappings from the history of input channels to the history of output channels. This is formalized as follows

> A mapping f from a complete partial order A into a complete partial order B is continuous iff, for any increasing chain a of A
> $f(\lim_A a) = \lim_B f(a)$

Kahn's usage of "increasing chain" to describe a was a subject of confusion. For this context, and from my understanding of the globally agreed upon definition of the word

'continuous,' "increasing chain" is assumed to suggest an infinite sequence of finite tuples of increasing length, and such that $a_1 \subset a_2 ... \in A$.

This property can be illustrated generally as:

$$rle(x) = y \tag{10}$$
$$|x| = |\mathbb{N}| \tag{11}$$
$$x_{s1} \subset x_{s2} ... \subset x_{s(n-1)} \subset x_{sn} ... \in x \tag{12}$$
$$x_{seqs} = (x_{s1}, x_{s2}, ...) \tag{13}$$
$$|x_{seqs}| = |\mathbb{N}| \tag{14}$$
$$z = (rle(x_{si}) \forall x_{si} \in x_{seqs}) \tag{15}$$
$$lim_{i \to \inf}(z_i) = y \tag{16}$$

These properties suggest that the application of run-length encoding is continuous upon application to infinite input sequences, and as such is a valid entity in a Kahn network. This formulation of the behavior of run length encoding allows its use in operating upon the nonterminating sequences arguably generated by real world hardware.

3.1 Caveats and Extensions

The logic of the previously described structure is contingent upon the operation of run length encoding to both finite and denumerably infinite sequences. The implementation of Listing 1, For finite inputs, always yields outputs of lesser cardinality than the input, as the terminating run produces no output two-tuple. This "lossy"-ness for finite sequences is undesirable, but a mechanism to adequately resolve this discrepency has not been adequately identified. For the purposes of this paper, the assumption that the terminating value of a finite input is repeated infinitely may be sufficient to resolve this. Alternatively, structuring Listing 1 to include a value not currently in the domain of the input demarcating termination, as per the formulation of null-terminated strings implemented on modern computers, would potentially resolve this discrepancy.

Additionally, (16) in the previously made mathematical statements is inadequately compelling to be regarded as proof of continuity, and future work may include complementing the assertions made with a more rigorous statement. The tautological assertion that the limit of an increasing chain is equivalent to the original denumerably infinite sequence is sensible, but inadequately grounded to be claimed as proof.

4. FUTURE WORK

Extending the pattern illustrating continuity set forth for run length encoding to run-length decoding, as well as the process of parsing itself, will be a topic of additional research. There is significant value to be gained from computational models with well-defined operation on infinite inputs, especially in the context of working with streams of data originating from our infinite world.

5. REFERENCES

[1] G. E. Allen and B. L. Evans. Real-time sonar beamforming on workstations using process networks and posix threads. *Signal Processing, IEEE Transactions on*, 48(3):921–926, 2000.

[2] T. R. Contributors. Rust programming language manual. *URL: http://static.rust-lang.org/doc/master/rust.html#tasks*, 2013.

[3] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *URL: http://swtch.com/ rsc/regexp/regexp1.html*, 2007.

[4] R. Cox. Regular expression matching in the wild. *URL: http://swtch.com/ rsc/regexp/regexp3.html*, 2010.

[5] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 35–44. IEEE, 2009.

[6] G. Kahn. The semantics of a simple language for parallel programming. 1974.

[7] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, 1995.

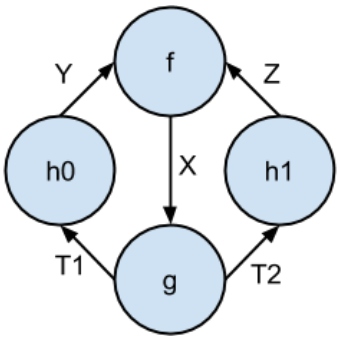
[8] Z. Vrba, P. Halvorsen, and C. Griwodz. Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors. In *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on*, pages 639–644. IEEE, 2009.

6. APPENDIX A. PROGRAM S

s.rs

```
use std::comm::{Port, Chan, stream};

fn f(U: Port<int>, V: Port<int>, W: Chan<int>) {
  loop {
    W.send(U.recv());
    W.send(V.recv());
  }
}

fn g(U: Port<int>, V: Chan<int>, W: Chan<int>) {
  loop {
    V.send(U.recv());
    W.send(U.recv());
  }
}

fn h(U: Port<int>, V: Chan<int>, INIT: int) {
  V.send(INIT);
  loop {
    V.send(U.recv());
  }
}

fn S_Init() {
  let (Xp,Xc): (Port<int>, Chan<int>) = stream();
  let (Yp,Yc): (Port<int>, Chan<int>) = stream();
  let (Zp,Zc): (Port<int>, Chan<int>) = stream();
  let (T1p,T1c): (Port<int>, Chan<int>) = stream();
  let (T2p,T2c): (Port<int>, Chan<int>) = stream();
  do spawn { f(Yp, Zp, Xc); }
  do spawn { g(Xp, T1c, T2c); }
  do spawn { h(T1p, Yc, 0); }
  do spawn { h(T2p, Zc, 1);}
}
```

Figure 1: Schematic of program S



7. APPENDIX B. PARSERS

Listing 2: original parsing scheme

```
enum morseSyms {
  dit,
  dah,
  interElementBreak,
  letterBreak,
  wordBreak,
}

fn validSymMorse(in: &rle::Run)
    -> Option(morseSyms) {
  match in.v {
    1 => { match in.ct {
      1 => Some(dit),
      3 => Some(dah),
      _ => None
      }},
    0 => { match in.ct {
      1 => Some(interElementBreak),
      3 => Some(letterBreak),
      7 => Some(wordBreak),
      _ =>     }},
    _ => None
  }
}

fn validCharMorse(in: &[morseSyms])
    -> Option(char){
  match in {
    &[dit, dit, dit] => Some('s'),
    &[dah, dah, dah] => Some('o'),
    &[wordBreak] => Some(' '),
    _ => None
  }
}
let encoded = rle(inputBitstream);
let symbols =
    encoded.filter_map(|&x| validSymMorse(x)).collect();
let characterSymbols: ~[&[morseSyms]] =
    symbols.split(|&x| x == letterBreak).collect();
let characters: ~[char] =
    characterSymbols.filter_map(|&x| validCharMorse(x)).collect();
```

Listing 3: KPN parser

```
pub fn morsed(U: Port<Run>, V: Chan<char>) {
  let mut morseSymbols: ~[morseSyms] = ~[];
  loop {
    let y = U.recv();
    match y.v {
      1 => match y.ct {
        1 => morseSymbols.push(dit),
        3 => morseSymbols.push(dah),
        _ => ()
      },
      0 => match y.ct {
        3 => { match morseSymbols {
          [dit,dit,dit] => V.send('s'),
          [dah,dah,dah] => V.send('o'),
          _ => ()
          };
          morseSymbols = ~[];},
        7 => V.send(' '),
        _ => ()
      },
      _ => ()
    }
  }
}
```

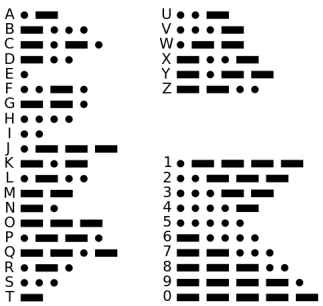


Figure 2: International Morse Code

In International Morse Code, the duration and order of a sequence of on and off pulses is used to encode the 26 English letters and 10 numerals. A dot, aka 'dot,' is a high pulse lasting one time unit. A dash, aka 'dah,' is a high pulse lasting three time units. A one time unit pause indicates the transition between successive symbols composing a letter. A three time unit pause indicates the transition between successive symbols, while a seven unit pause composes a wordbreak, or space.