Aerospace Technology Congress, 8-9 October 2019, Stockholm, Sweden
Swedish Society of Aeronautics and Astronautics (FTF)

Lempel-Ziv-Markov Chain Algorithm Modeling using Models of Computation and ForSyDe

Augusto Y. Horita[1], Ricardo Bonna[1], Denis S. Loubach[2], Ingo Sander[3], and Ingemar Söderquist[4]

E-mail: ahorita@fem.unicamp.br; rbonna@fem.unicamp.br; dloubach@ita.br; ingo@kth.se; ingemar.soderquist@saabgroup.com
[1]Advanced Computing, Control & Embedded Systems Lab, University of Campinas – UNICAMP, Campinas, SP, Brazil - 13083-860
[2]Department of Computer Systems, Computer Science Division, Aeronautics Institute of Technology – ITA, São José dos Campos, SP, Brazil - 12228-900
[3]Division of Electronics/School of EECS, KTH Royal Institute of Technology, SE-164 40, Kista, Sweden
[4]Business Area Aeronautics, Saab AB, Linköping, Sweden

Abstract

The data link is considered a critical function of modern aircraft, responsible for exchanging information to the ground and communicating to other aircraft. Nowadays, the increasing amount of exchanged data and information brings the need for network usage optimization. In this sense, data compression is considered a key approach to make data packages size smaller. Regarding the fact that avionics systems are safety-critical, it is fundamental not losing data nor performance during the compression procedures. In this context, manufacturers and regulatory agencies usually follow DO-178C guidance. Targeting model-based embedded design guidelines, DO-178C includes a supplement document, named DO-331. In this paper, we describe a widely used data compression algorithm, the Lempel-Ziv-Markov Chain algorithm (LZMA). Regarding formal model-based design, we argue that the synchronous dataflow model of computation captures the algorithm behavior more directly. The Formal System Design (ForSyDe) methodology is used to model the LZMA.

Keywords: data compression algorithm, avionics data link, DO-178C, DO-331, formal models of computation, synchronous dataflow.

1 Introduction

A wide range of aircraft functions with different safety requirements is present in the avionic system of today's modern aircraft. One of these aircraft functions is the *data link*. It is responsible for exchanging information to the ground, besides communicating to other aircraft during flight.

As the number of aircraft functions increases, the complexity of avionics systems exponentially grows. The number of processors, transducers and exchanged data and information also increase. This brings the need for network usage optimization. In this sense, *data compression* is considered a key approach to making data packages size smaller.

Considering that the avionics systems are safety-critical, it is fundamental not losing data nor performance during the compression procedures [1]. In this scenario, manufacturers and regulatory agencies follow the DO-178C [2] guidance. That document is developed and maintained by the Radio Technical Commission for Aeronautics. DO-178C aims to ensure that software development for avionics systems is dependable, safe, and meet the specified requirements.

Because of this, formal models of computation (MoC) have been used to *model*, *simulate* and *verify* algorithms in the system design and implementation phases, considering the embedded systems area. DO-178C includes a supplement document describing model-based design guidelines, named DO-331 [3].

Given this context, we address in this paper the formal modeling and simulation of a widely used data compression algorithm, named Lempel-Ziv-Markov Chain algorithm (LZMA). It is included as a CPU benchmark by the Standard Performance Evaluation Corporation (SPEC) [4].

Regarding formal model-based design, we argue that the synchronous dataflow (SDF) model of computation captures the algorithm behavior more directly. The LZMA intends to generate a compressed file based on the processing of a general data stream input. The Formal System Design (ForSyDe) methodology [5] is used to model the LZMA in our paper.

2 Background

This section presents the concepts used along with this paper including *models of computation* (MoC), *synchronous dataflow* (SDF) MoC, and the *Lempel-Ziv-Markov Chain* algorithm (LZMA).

2.1 Model of Computation (MoC)

An MoC represents an abstraction of a real computing device [6], which may have different behaviors and applications. It includes the relevant characteristics and properties for that particular model. Thus, different MoCs are used for modeling different systems depending on their behavior.

MoCs can be defined as an abstract rules collection stating the semantics of execution and concurrency in computational systems.

In this context, the *tagged signal model* (TSM) is also present. It is introduced as a meta-model, or even a framework, defining systems as compositions of processes acting on signals [7].

A *signal* is a set of events $e_i = (t_i, v_i)$, which are elementary units of information composed by a tag $t_i \in T$, and a value $v_i \in V$. A signal can be viewed as a subset of $T \times V$. A *process* P is a set of possible behaviors acting on a signal. The set of output signals S^O is given by the intersection between the input signals S^I and the process: $S^O = S^I \cap P$. A process is *functional* when there is a single value mapping $f : S^I \rightarrow S^O$ which describes it. Therefore, a functional process has either one behavior or no behavior at all.

TSM divides MoCs basically into *timed* and *untimed*. In a *timed* MoC, the set of tags T is *totally ordered*, *i.e.* one can order every event included in the model based on its tag. On the other hand, in an *untimed* MoC, the set of tags T is *partially ordered*, *i.e.* only local groups of events can be ordered based on its tag, *e.g.* the ones belonging to the same signal.

2.2 Synchronous Dataflow (SDF) MoC

A subclass included in the untimed MoCs comprehends the *dataflows*, which are directed graphs where each *node* represents a process and each *arc* represents a signal path. When a process is activated, *i.e. fired*, it consumes a certain amount of data, denominated *tokens*, from its input ports and generates another amount of tokens for its output ports. The amount of tokens consumed and produced by the ports in a single activation cycle is denominated *token rate* [8].

In SDF, the token rates are fixed and predefined, represented by a natural number associated with each input and output ports. An actor can fire only if the input signal paths have enough tokens to supply the amount needed by all input ports of the actor. As a consequence, no signal path can have a negative amount of tokens.

The predefined and fixed token rates of SDF actors allows efficient modeling of systems like signal processing or finding a static schedule for single and multi-processor implementation and also buffer size definitions [8].

2.3 Data Compression Algorithms

Data compression has been used to optimize storage and communication buses. One of the compression techniques is based on dictionaries, which consists of saving strings of a previously read input stream interval, composing a dictionary. When new groups of symbols are being read in, the algorithm searches through the dictionary, then the matches are encoded as pointers and sent as the output.

Abraham Lempel and Jacob Ziv have presented their first dictionary-based compression method in 1977 [9], which is referred to as LZ77. That method limits the previously read interval to a determined size window, which follows the data processing, creating the concept of *sliding window encoding* (SWE), which makes the algorithm simple and faster [10]. In 1978, LZ78 was presented [11]. The difference is that its dictionary is composed of all the symbol strings from previously read stream and it builds a single character at a time, making this algorithm longer but with a higher compression rate.

The algorithm analyzed in this paper, LZMA, was first used in 7z file format [12] and is presented as a benchmark by SPEC [4], comparing its implementation using different architectures.

2.3.1 Lempel-Ziv-Markov Chain Algorithm (LZMA)

LZMA was created as an LZ77 optimization, providing higher compression rate and fast decompression, with lower memory requirements [13]. It is open source and implemented in several different programming languages, including C, and Java, as an SDK [12].

LZMA can be divided into two steps, although an additional filtering step can be added before LZMA, aiming the optimization of the compression algorithm, as illustrated in Fig. 1.

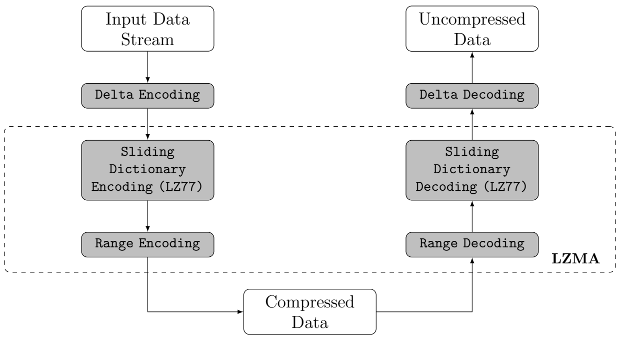

Figure 1: LZMA compression and decompression block diagram, based on [10].

Step$_1$ *Delta encoding* – this step consists of an optional data filtering that encodes the input stream outputting, for each byte, a byte representing the difference between the current and the previous bytes. The first byte output is itself. This process makes the sliding window more efficient;

Step$_2$ *Sliding dictionary encoding* – this step is based on the LZ77 algorithm, although it supports larger dictionaries.

Optimized search algorithms were implemented to perform faster searches in these dictionaries, named *hashed chain* and *binary trees* [13]. The output is, as in LZ77, a sequence of triplets, composed by the *distance* from the string in the look-ahead buffer to its match in the search buffer, the *length* of the string, and the next input *symbol*; and

$Step_3$ *Range encoding* – this step is context-based. The compressed range in each iteration is estimated based on probabilistic algorithms and can form a set of predefined types of packages depending on the input range size. This was first presented by [14]. In LZMA, the compression adopted context is the output of $Step_2$ in each firing cycle.

3 Lempel-Ziv-Markov Chain Modeling

This section presents the LZMA modeling based on the SDF MoC. Next, a brief model description is presented, followed by the model design using ForSyDe.

3.1 Model Description

The LZMA compression has as input a data stream, which is processed by a sequence of defined steps, outputting a compressed and encoded byte stream. Such behavior can be modeled using a dataflow MoC. Towards simple modeling, our paper presents the LZMA compression model based on SDF MoC, considering the decompression to be a similar process.

LZMA has a set of parameters that are user-defined at the compression beginning. These parameters configure the algorithm behavior, data structures, and performance. For an initial formal modeling purpose, this paper considers some specific configurations. The following introduces a list of assumptions taken into consideration to model the simplified LZMA version.

A_1 the absence of the delta encoding filter;

A_2 the sliding window encoder step is changed to use a similar algorithm, *i.e.* LZ78 sliding window encoder [11] was adopted instead; and

A_3 a fixed probability range encoder.

Figure 2 illustrates the LZMA modeled as a SDF functional graph.

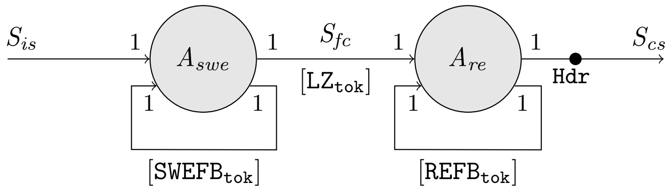

Figure 2: LZMA SDF MoC graph

The model blocks and communication paths are defined as follows:

- A_{swe} – Sliding window encoding block. It encodes the input stream based on a dictionary structure, outputting a literal, a match or an absent value in each process firing;
- A_{re} – Range encoding block. It performs a bit-wise compression of the literals and matches, generating a byte for each one, resulting in the final compressed stream;
- S_{is} – Input stream. It comprehends the input to be compressed;
- S_{fc} – Compressed stream first step. It contains the tokens already compressed using the sliding window encoding method;
- S_{cs} – Compressed stream. It represents the final LZMA output;
- $\mathtt{SWEFB_{tok}}$ – Sliding window encoding feedback token. It is composed of the dictionary data structure, updated at each A_{swe} firing;
- $\mathtt{REFB_{tok}}$ – Range encoding feedback token. It is composed of the considered range, lower limit and encoded character cache memory, updated at each A_{re} firing;
- $\mathtt{LZ_{tok}}$ – Sliding window encoder output token. It can have an absent value or a tuple, depending on the A_{swe} read input token. In the latter case, the tuple is composed by a read character and another entry. If the read character is not in the dictionary, the first tuple entry is an absent value, and the token represents a *Literal*. Otherwise, the first entry represents the number of characters that was repeated inside the sliding window, and the token represents a *Match*; and
- `Hdr` – Initial bytes contained in the output compressed stream. It comprehends the configuration setup, besides the dictionary and uncompressed input stream sizes.

3.2 Modeling supported by ForSyDe

The ForSyDe framework was chosen to be used in the present paper based on the frameworks comparison presented in [15]. In view of that, we use ForSyDe to model LZMA as described in the previous section.

In ForSyDe, the SDF MoC actors, *i.e.* blocks presented in last section, are modeled as *processes*, and the communication paths as *signals*.

Listing 1 presents the model processes definitions, using the process constructors from the ForSyDe SDF library. For brevity, this paper only presents the main processes and signals models and definitions. However, the full functional implementation of the LZMA model can be found in [16].

Regarding the sliding window encoding, the A_{swe} actor was modeled as `lzA`, using the ForSyDe SDF process constructor `actor22SDF`, which indicates that the process has 2 inputs and 2 outputs. The $\mathtt{SWEFB_{tok}}$ dictionary was modeled as `fb`. The LZMA input S_{is} was represented by the `sIs` signal, and the S_{fc} by the `sFc` signal.

When modeling the range encoding step, the actor A_{re} was represented as `rgA` based on the process constructor actor22SDF. The $\mathtt{REFB}_{\mathtt{tok}}$ range encoding variables tuple was modeled as sFb. The output compressed stream S_{cs} was modeled as sCs, and the initial header Hdr as initHdr.

Listing 1: LZMA code snippet in ForSyDe SDF/Haskell

```
-- LZ (Sliding Window) Encoding actor definition
-- Input is the lzma input stream Sis
-- Output is the first step compressed stream Sfc
lzA :: Signal Char -> Signal (Maybe (Maybe Int, Char))
lzA Sis = Sfc
  where (Sfc, fb) = actor22SDF (1,1) (1,1) lzF Sis fb'
        fb' = delaySDF [([],"",0)] fb

-- Range Encoding actor definition.
-- Input is the first step compressed stream Sfc
-- Output is the compressed LZMA stream output Scs
rgA :: Signal (Maybe (Maybe Int, Char))
        -> Signal [Char]
rgA sFc = sCs'
  where (sCs,sFb) = actor22SDF (1,1) (1,1) rangeFunc
    sFc sFb'
        sCs' = delaySDF initHdr sCs
        sFb' = delaySDF [(rangeInit,0, chr 0)] sFb
        initHdr = [([dictsize] ++ [inpLen])]
        inpLen = (intToDigit (lengthS sFc))
```

Listing 2 presents how the simplified LZMA *process network* lzmaSdf is assembled to compose the algorithm completely written using ForSyDe MoC. The range encoding process rgA is executed having the output of sliding window process lzA as input.

Listing 2: LZMA process netword in ForSyDe SDF/Haskell

```
module Lzmasdf (
    lzmaSdf
  ) where

import ForSyDe.Shallow
import Rangesdf
import LZsdf
import Data.Char

-- Simplified LZMA process network
lzmaSdf :: Signal Char -> Signal [Char]
lzmaSdf sLzmaIs = sLzmaOut
    where sLzmaOut = rgA (lzA sLzmaIs)
```

4 Summary

This paper presented a simplified Lempel-Ziv-Markov Chain algorithm (LZMA) modeling based on the synchronous dataflow (SDF) model of computation (MoC) using the ForSyDe framework. Some configurations and behaviors assumptions were adopted towards the definition of actors ports fixed token rates. Those assumptions allowed for a first simple LZMA formal modeling and simulation.

As future work, we plan to implement LZMA based on the scenario-aware dataflow (SADF) modeling and simulation framework introduced in [17], which supports variable token rates, resulting in a more advanced LZMA model.

References

[1] Paul Berthier, Corentin Bresteau, and José Fernandez. On the security of aircraft communication networks. In Gregorio D'Agostino and Antonio Scala, editors, *Critical Information Infrastructures Security*, pages 266–269, Cham, 2018. Springer International Publishing.

[2] Radio Technical Commission for Aeronautics - RTCA. *DO-178C - Software Considerations in Airborne Systems and Equipment Certification*, 2012.

[3] Radio Technical Commission for Aeronautics - RTCA. *DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A*, 2011.

[4] Standard Performance Evaluation Corporation (SPEC). 657.xz_s spec cpu 2017 benchmark description. http:/www.spec.org/cpu2017/Docs/benchmarks/657.xz_s.html, 2019.

[5] Ingo Sander. The forsyde methodology. In *Swedish System-on-Chip Conference*, 2002.

[6] Axel Jantsch. Models of embedded computation., 2005. Embedded Systems Handbook, chapter Models of Embedded Computation. CRC Press.

[7] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, Dec 1998.

[8] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[9] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[10] Epiphany Jebamalar Leavline. Hardware implementation of lzma data compression algorithm. In *International Journal of Applied Information Systems (IJAIS)*, 2013.

[11] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sep. 1978.

[12] Igor Pavlov. 7z format. http://www.7-zip.org/7z.html, 2019.

[13] David Salomon. *Data Compression: The Complete Reference*. Springer, 2007. With contributions by Giovanni Motta and David Bryant.

[14] G. Nigel Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference*, 1979.

[15] Augusto Y. Horita, Ricardo Bonna, and Denis S. Loubach. Analysis and comparison of frameworks supporting formal system development based on models of computation. *Springer - Advances in Intelligent Systems and Computing*, 2019.

[16] A. Y. Horita. Forsyde simplified lzma sdf model source code. https://github.com/AugustoHorita/LzmaForSyDeSdf, 2019.

[17] Ricardo Bonna, Denis S. Loubach, George Ungureanu, and Ingo Sander. Modeling and simulation of dynamic applications using scenario-aware dataflow. *ACM Transactions on Design Automation of Electronic Systems*, 2019.