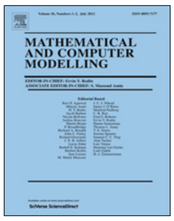
Mathematical and Computer Modelling 58 (2013) 1548–1562

Contents lists available at SciVerse ScienceDirect

Mathematical and Computer Modelling

journal homepage: www.elsevier.com/locate/mcm



Upgraded Petri net model and analysis of adaptive and static arithmetic coding

Perica Štrbac[a], Gradimir V. Milovanović[b,*]

[a] Faculty of Computer Sciences, Megatrend University, Bulevar Umetnosti 29, 11070 Novi Beograd, Serbia

[b] Mathematical Institute of the Serbian Academy of Sciences and Arts, Knez Mihailova 36, p.p. 367, 11001 Beograd, Serbia

ARTICLE INFO

Article history:
Received 25 June 2011
Accepted 1 June 2013

Keywords:
Arithmetic coding
Upgraded Petri nets
Data compression
Source coding

ABSTRACT

In this paper, we analyze several adaptive and static data models of arithmetic compression. These models are represented by using Upgraded Petri net as our original class of the Petri nets. After the iterative processes of modeling, simulation and analysis, the models are transformed into an application. The models refer to one-pass and two-pass arithmetic coding where a set of symbols refers to bytes or nibbles. The frequency of a symbol is represented as an unsigned 32-bit integer. Original software for modeling and simulations of Upgraded Petri net, PeM (Petri Net Manager) is developed and used for all models described in this paper. All models are observed in the experiments by using a created application over a standard set of files. Experimental results are presented and compared.

1. Introduction

Upgraded Petri nets are a formal mathematical apparatus which enable modeling, simulation and process analysis [1] as does a Petri net [2]. We developed Upgraded Petri nets as a new class of Petri nets. These are graphical and mathematical modeling tools applicable to many systems. Upgraded Petri nets are used for describing and studying information processing systems that are characterized as being concurrent, asynchronous, synchronous, distributed, parallel, deterministic, nondeterministic, and/or stochastic as is the Petri net. As a graphical tool, Upgraded Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. Transition firing level, tokens and transition functions are used in these nets to simulate the dynamic and concurrent activities of systems. They enable interactive monitoring of process operations and its gradual improvement from the initial phase, all the way to the final version. We use the Upgraded Petri nets (UPN) which we developed in order to enable use of Petri nets for hardware modeling, as well as to provide modeling at register level [1,3]. All models in the paper are created by using UPN. The hierarchical structure of a UPN gives many possibilities for abstraction. This feature of the UPN provides the model implementation consisting at the same time of elaborate pieces essential for the analysis at a certain level, and also of some general pieces whose details are irrelevant for the analysis at the given level of abstraction [1]. We are using Petri net based simulation via an originally developed Petri nets Manager (PeM) software suite. The PeM supports the formal theory of the Petri net and also, the formal theory of the Upgraded Petri net. It is used for modeling, simulation and analysis of all models shown in the paper. The PeM concurrently fires enabled transitions with respect to the rules of the UPN execution which is shown in the next section.

Arithmetic coding maps an input string of data symbols to a code string in such a way that the original data can be recovered from the code string, i.e. it is a form of lossless data compression. Arithmetic coding has been widely used in data compression [4–7]. This type of coding can encode a sequence from the source at a rate very close to the entropy rate.

[*] Corresponding author. Tel.: +381 26 30 170.
E-mail addresses: pstrbac@megatrend.edu.rs (P. Štrbac), gvm@mi.sanu.ac.rs, gvm@sbb.rs (G.V. Milovanović).

0895-7177/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.mcm.2013.06.001

Arithmetic coding can be easily used in conjunction with sophisticated probability models. The disadvantage of arithmetic coding is its relatively high computational complexity.

In this paper, we consider adaptive and static data models of arithmetic coding. In the work we make an Upgraded Petri nets model, conduct a simulation, and after software implementation of the model we finally conduct analysis of several data models of adaptive and static arithmetic coding over the standard set of files.

2. Upgraded Petri nets

In this section, we present Upgraded Petri nets [1,3] that we developed in order to enable use of Petri nets for easy modeling of hardware, as well as to provide suitable modeling at register level. Software suite PeM supports: graphic modeling of the Upgraded Petri net, execution of the Upgraded Petri net and reachability tree generation.

2.1. An upgraded Petri net formal theory

Let $\mathbb{N}_0$ be the set of non-negative integers, and P and T be disjoint finite nonempty sets of places $p_i, i = 1, \dots, n$, and transitions $t_j, j = 1, \dots, m$, respectively, i.e.,

$$P = \{p_1, p_2, \dots, p_n\}, \qquad T = \{t_1, t_2, \dots, t_m\}, \quad n, m > 0.$$

A formal theory of Upgraded Petri nets is based on functions as introduced in [1]. An Upgraded Petri net is a 9-tuple

$$C = (P, T, F, B, \mu, \theta, TF, TFL, PAF),$$

where

$F : T \times P \to \mathbb{N}_0$ – Input Function,

$B : T \times P \to \mathbb{N}_0$ – Output Function,

$\mu : P \to \mathbb{N}_0$ – Marking Function,

$\theta : T \times \mathbb{N}_0 \to [0, 1]$ – Timing Function,

$TF : T \to \mathrm{A}$ – Transition Function,

$TFL : T \to \mathbb{N}_0$ – Transition Firing Level,

$PAF : P \to (x, y)$ – Place Attributes Function.

The input function assigns a non-negative number to an ordered pair $(t_i, p_j) \in T \times P$. The assigned non-negative integer defines how many times the place p_j is an input with respect to the transition t_i.

The set of places which are input with respect to the transition t_j is presented by ${}^*t_j = \{p_i \in P : F(t_j, p_i) > 0\}$. For a presentation of the place $p_i \in {}^*t_j$ which has the standard input with respect to the t_j, we use the notation ${}^*t_j^S$, and $F^S(t_j, p_i)$ we use for such an input function. For places $p_i \in {}^*t_j$ with inhibitor input with respect to the t_j transition, the notation ${}^*t_j^I$ will be used and $F^I(t_j, p_i)$ for the input function.

The output function maps an ordered pair $(t_i, p_j) \in T \times P$ to the set of non-negative integers. The assigned non-negative integer shows how many times the place p_i is an output with respect to the t_i transition. A set of places which are output with respect to the t_j transition is presented as follows

$$t_j{}^* = \{p_i \in P : B(t_j, p_i) > 0\}.$$

The marking function assigns a non-negative integer to the p_i place. The marking function can be defined as n-dimensional vector (marking): $\mu = (\mu_1, \mu_2, \dots, \mu_n)$, where $n = |P|$. Instead of μ_i it can be used the notation $\mu(p_i)$.

The timing function θ assigns the probability $\lambda_{ij} \in [0, 1]$ to an ordered pair $(t_i, j) \in T \times \mathbb{N}_0$, i.e., $\lambda_{ij} = \theta(t_i, j)$.

The transition function gives an operation $\alpha_j \in \mathrm{A}$ to the t_j transition. Here, A denotes the set of operations which can be assigned to the transition.

The firing level function of the transition gives a non-negative integer to the transition t_j. If this number does not equal zero, it shows the number of $p_i \in {}^*t_j$ places takes part in the transition firing, and if this number equals zero, then all the places $p_i \in {}^*t_j$ affect the t_j transition firing.

The place attributes function assigns an ordered pair (x, y) to the place p_i. The x component is a real number, called the x attribute, and y is a non-negative integer called the y attribute (i.e., $x \in \mathbb{R}$, $y \in \mathbb{N}_0$). Over the x attribute belonging to the $p_i \in {}^*t_j$ places, the α_j operation assigned to the t_j transition executes, where the order of operands in the operation α_j is defined by the y attributes which belong to the p_i place in accordance with the *TFL* function taking part in the transition firing.

An Operation Assigned to a Transition — the function *TF* assigns to a transition t_j one operation. This operation can be an arithmetical operation, logical operation or a file operation [1]. Inside the suite PeM, a file which is a target of file operation function has an `*.mem` extension. This `*.mem` file is a text file and it is used for the simulation of computer system memory. One line inside the `*.mem` file refers to the context of one memory location of a computer system that we are modeling.

An arithmetical operation $\alpha_j \in \mathrm{A}$, which is assigned to the transition $t_j \in T$, uses attributes x which belong to the places $p_i \in {}^*t_j$ as operands of that operation. A result of an arithmetical operation $\alpha_j \in \mathrm{A}$ will be placed into the attributes x which belong to the places $p_i \in t_j^*$. The order of an operand (i.e. order of attributes x which belong to the places $p_i \in {}^*t_j$) in an arithmetical operation $\alpha_j \in A$ is defined by attributes y which belong to the places $p_i \in {}^*t_j$.

A logical operation $\alpha_j \in A$ which is assigned to the transition $t_j \in T$, uses attributes x which belong to the places $p_i \in {}^*t_j$ as operands of that operation. If a result of the logical operation $\alpha_j \in A$ is logical *false* the transition $t_j \in T$ is disabled and will stay in that state until the result of this logical operation $\alpha_j \in A$ becomes logical *true*. The order of an operand (i.e., order of attributes x which belong to the places $p_i \in {}^*t_j$) in a logical operation $\alpha_j \in A$ is defined by attributes y which belong to the places $p_i \in {}^*t_j$.

A file operation $\alpha_j \in A$ which is assigned to the transition $t_j \in T$ performs over the context of a file whose extension is equal to `*.mem`. A File Operation $\alpha_j \in A$ addresses the context of a `*.mem` by using attribute x which belongs to the places $p_i \in {}^*t_j$. A result of this operation $\alpha_j \in A$ changes the value of attributes x which belong to the places $p_i \in t_j{}^*$. The result also can change attributes y which belong to the places $p_i \in t_j{}^*$, or can change the context of addressed line into the `*.mem` file.

According to a UPN Graph representation we can say the following. The Upgraded Petri net can be represented via formal mathematical apparatus or graphically. Namely, a UPN is represented by a bipartite multigraph [1] in the same way as a Petri net.

2.2. An upgraded Petri net executing

An Upgraded Petri net executing represents a change of system state from the current state to the next state. This migration from one state to the other one is triggered by firing of the transitions. By UPN executing: the marking vector can be changed, the contents of `*.mem` file can be changed, and attributes which belong to the places $p_i \in t_j{}^*$ of enabled transition t_j can be changed.

A transition $t_j \in T$ can be enabled at the moment $k \in \mathbb{N}_0$ in an Upgraded Petri net [1]: $C = (P, T, F, B, \mu, \theta, TF, TFL, PAF)$ if the next three conditions are satisfied:

1° If the timing function $\lambda_{jk} = \theta(t_j, k) > 0$;
2° If $TFL(t_j) > 0$ then $(\#p_i(S)) + (\#p_i(I)) = TFL(t_j)$, and if $TFL(t_j) = 0$ then $(\#p_i(S)) + (\#p_i(I)) = |{}^*t_j|$, where $\#p_i(S)$ is a number of places $p_i \in {}^*t_j^S$ such that $\mu(p_i) \geq F^S(t_j, p_i)$, and $\#p_i(I)$ represents a number of places $p_i \in {}^*t_j^I$ for which $\mu(p_i) = 0$;
3° If a logical operation $\alpha_j \in A$ is assigned to the transition t_j, then the result of the operation α_j must be equal to true.

A marking vector μ will be changed to a new marking vector μ' by firing of transitions t_j, where

$$\mu'(p_i) = \begin{cases} \mu(p_i) - F(t_j, p_i) + B(t_j, p_i), & \text{if } p_i \in {}^*t_j^S, \\ \mu(p_i) + B(t_j, p_i), & \text{if } p_i \in {}^*t_j^I. \end{cases}$$

By firing of the transition t_j an arithmetic operation is executed or a file operation is executed with respect to the operation that is assigned to t_j by function $TF(t_j)$.

A logical operation which is assigned to the transition t_j by function $TF(t_j)$ will be executed if the conditions 1° and 2° related to t_j are equal to true.

A conflict in an Upgraded Petri net influences UPN executing. A conflict in a UPN is the same as the conflict in a Petri net.

A UPN reachability tree graphically represents all possible marking vectors which can occur during a UPN execution for a given initial marking. The reachability tree shows all states the model can reach from the initial state. The UPN reachability tree is the same as the Petri Net reachability tree.

A UPN executing refers to a concurrent firing of the enabled transitions. A UPN execution generates a UPN flammability tree. This tree is a tree where a node of the tree is a set of the transitions which are enabled at the same time. If there is the same node as the current node in the flammability tree then generating of the flammability tree will be stopped. There are four types of nodes in a flammability tree: root node, double node, dead node, and inner node.

3. Arithmetic coding

In this section we show some definitions of the basic terms used in the paper and explain how arithmetic coding based on fixed arithmetic works by using integer numbers [8–11]. We use an arithmetic coder in which the range allocated to each symbol is a single contiguous interval and no permutations of ranges are applied [10,12]. There are several approaches to the symbol sets and multiple tables in arithmetic coding [13,14] so we use one symbol set and one coding table.

3.1. Definitions of some basic terms

The *alphabet* A_L is a finite, nonempty ordered set. We use mark A_L because mark A denotes a set of operations which can be assigned to the transition in Upgraded Petri nets. The elements $\{a_1, \ldots, a_n\}$ of an alphabet are called *symbols*. These elements are distinctly ordered. The *cardinality* of an alphabet A_L will be referred to as $|A_L|$.

We consider a *sequence* $S = s_1, s_2, \ldots$ of symbols s_i from the alphabet A_L, where $|S| < \infty$.

Let $S = (s_1, \ldots, s_m)$ be a finite-length sequence with $|S| = m$ over $A_L = \{a_1, \ldots, a_n\}$.

Let $|S(a_i)|$ refer to the frequency of a_i in S.

The probability of a_i in S is

$$P(a_i) = \frac{|S(a_i)|}{m}.$$

The lower bound *Low* is the sum of frequencies of all lower symbols than the current symbol. The lower symbols are the symbols which have an index less than the current symbol.

$$Low = \sum_{i=1}^{s-1} Fc_i.$$

The mark s denotes an index of the current symbol while the mark Fc_i is the frequency count of the symbol indexed by i.

The upper bound *High* is a sum of the lower bound and the frequency of the current symbol,

$$High = \sum_{i=1}^{s} Fc_i = Low + Fc_s.$$

Total frequencies

$$T_f = \sum_{i=1}^{m} Fc_i$$

is a sum of all frequencies of the symbols in S.

3.2. Arithmetic coding algorithm

We introduce some basic terms used in arithmetic coding data compression and decompression that will be used in the paper. In this paper we treat this data as binary input stream. A group of such input bits is referred to as a symbol. There are two types of symbol that we consider in this work: byte (8 bits) and nibble (4 bits).

The arithmetic coding algorithm works sequentially over the input data stream. We use fixed precision arithmetic, interval expansion and bit-stuffing [4,15–19]. The main idea is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown part of the final interval. Witten, Neal, and Cleary [4] add a mechanism for preventing the current interval from shrinking too much when the endpoints are close to the half of the start interval but straddle this half. In that case we do not yet know the next output bit, but we do know that whatever it is, the following bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically by about half of the start interval. This procedure may be repeated any number of times, so the current interval size is always longer than the quarter of the *start interval*.

In this paper the *start interval*, i.e., $[Low, High)$ is initialized to $[0, 2^{31} - 1)$. For each symbol from the input stream two steps will be performed. The first one is subdividing of the current interval into subintervals, one for each symbol from the alphabet. These subintervals are proportional to the frequency of the symbol. The second step is selecting the subinterval regarding the current symbol from the input stream and setting this interval as the new current interval. Finally we will save leading bits to distinguish the new current interval from further intervals which will be result of interval expansion. The new subinterval is calculated as follows

$$raster = \frac{High - Low + 1}{T_f}, \tag{3.1}$$

$$High = Low + raster \cdot \left(\sum_{i=1}^{s} Fc_i \right) - 1, \tag{3.2}$$

$$Low = Low + raster \cdot \sum_{i=1}^{s-1} Fc_i, \tag{3.3}$$

where *raster* is a step size of the old subinterval $[Low, High)$ divided by the total count of frequencies T_f.

When we encode more and more symbols, the *Low* and the *High* converge more and more and in one moment these two values coincide and further encoding will be impossible. To avoid this problem an interval expansion is used. This expansion takes place after the selection of the subinterval. There are four cases of interval expansion process.

The first case is where the subinterval lies entirely within the first half of the *start interval*. In this case we save a zero bit and any 1s left over from the previous symbol to the output stream, then we will double subinterval $[Low, High)$ toward the right. This is $e1$ scaling.

The second case is where the subinterval lies entirely within the second half of the *start interval*. In this case we save 1 and any 0s left over from previous symbol to the output stream, then we will double subinterval $[Low, High)$ toward the left. This is $e2$ scaling.

The third case is where the subinterval lies entirely within the first quarter and the third quarter of the *start interval*. In this case we keep track of this fact for future output, then we will expand subinterval $[Low, High)$ in both directions away from the midpoint of the *start interval*. This is $e3$ scaling.

The fourth case includes all other situations and no expansion will happen. The expansion process repeats until the fourth case happens.

In this paper all symbols from S have the initial frequency set to 1. Also we use an ESC symbol to represent the end of coding, so at the start of coding $|S| = 256 + 1$ if a type of symbol is byte or $|S| = 16 + 1$ if a type of symbol is nibble.

When we decode an input stream, the main goal is to determine the symbol and update the bounds accordingly. The first task is to determine the interval that contains the symbol, then to calculate the code value of the symbol. We use a mark B to denote a buffer which contains an encoded stream. The procedure of decoding is to calculate formula (3.1), then to calculate

$$value = \frac{B - Low}{raster},$$

then an encoded symbol will be determined by comparing the *value* to the cumulative frequency count intervals. When the proper interval is found the boundaries will be updated according to the formulas (3.2) and (3.3).

4. Upgraded Petri net arithmetic coding models

In this section we explain Upgrade Petri net arithmetic coding models which we developed and used in this paper. The data model used in arithmetic coding can be adaptive or static. In this paper we use both adaptive and static data models.

In this section we will use some abbreviations in our UPN models as follows: B refers to a buffer where the input data stream will be loaded, Hf refers to a half of the start interval, FQ refers to a first quarter of the start interval, SC refers to an $e3$ scaling counter, L refers to Low, H refers to $High$, ESC refers to the symbol which ends arithmetic coding, and LB refers to a bit which is loaded from the input stream.

4.1. Adaptive data model

Adaptive arithmetic coding is a one-pass procedure where the coder builds up the statistical model while coding the input data.

Fig. 1 (left) shows a UPN model of arithmetic coding compression and decompression at high representation level. Initial marking $\mu(p_1) = \mu(p_2) = 1$ determines a source file which will be compressed or decompressed and the compression or decompression mode.

The model refers to two modes of compression and decompression: an adaptive and static one. Fig. 1 (left) shows that an adaptive encode is selected, i.e., the transition t_1 is enabled. This UPN model has four possible sequences of transition firing. The first sequence of transition firing is $\{t_1\} - \{t_5\} - \{t_7\}$ and refers to an adaptive encode. The second sequence is $\{t_2\} - \{t_6\} - \{t_7\}$ and refers to a static encode. Transition t_7 refers to the final encode for both adaptive and static encodes. An event *final encode* will be explained later. The third sequence is $\{t_3\} - \{t_8\} - \{t_9\}$ and refers to an adaptive decode, while the fourth sequence $\{t_4\} - \{t_8\} - \{t_{10}\}$ refers to a static decode. Transition t_8 refers to a fill buffer before decode process started.

Fig. 1 (right) shows a UPN model of adaptive arithmetic coding compression at an input stream encode level of representation. Initial marking $\mu(p_1) = 1$ and two enabled transitions t_1 and t_6 refer to two possible cases.

The first case is that there is a symbol in the input stream which we want to encode. A sequence of firing transitions $\{t_1\} - \{t_2\} - \{t_3\} - \{t_4\}$ occurs. This sequence refers to symbol loaded, then, update table of frequencies, then encode the symbol, then increment a total of encoded symbols and finally check again if there is there another symbol in input steam.

The second case is that there are no more symbols in the input stream. The sequence of firing transitions $\{t_6\} - \{t_5\}$ encodes ESC symbol and checks Low value.

If this value is less than the first quarter of the start interval, a sequence $\{t_7\} - \{t_{11}\} - \{t_9\} - \{t_{12}\}$ occurs. This sequence saves bit 0, then saves bit 1 $SC + 1$ times, then saves bit 0 and finally by firing of the transition t_{12} the output stream is populated by zeros up to the next byte.

On the other side, if the *Low* value is greater than or equal to the first quarter of the coding interval a sequence $\{t_8\} - \{t_{10}\}$ occurs. This sequence saves bit 1. At last, transition t_{12} is enabled. By firing of this transition the output stream is populated by zeros up to the next byte.

All savings we mentioned in the sections refer to the output stream.

Fig. 2 (left) shows a UPN model of adaptive arithmetic coding compression at a symbol encode level of representation. Initial marking $\mu(p_1) = 1$ determines that a sequence of transition firing $\{t_1\} - \{t_2\} - \{t_3\}$ occurs. This sequence calculates the *Low* value (denoted as an L mark on the figure), then calculated the *High* value (denoted as an H mark on the figure) and checks conditions for $e1$ scaling, $e2$ scaling and for $e3$ scaling update.

The condition for $e1$ scaling is that *High* is less than half of the coding interval value. The condition for $e2$ scaling is that Low is greater than or equal to a half of the coding interval. The condition for $e3$ scaling update is that *Low* is greater than or equal to the first quarter of the coding interval and that *High* is less than the third quarter of the interval value.

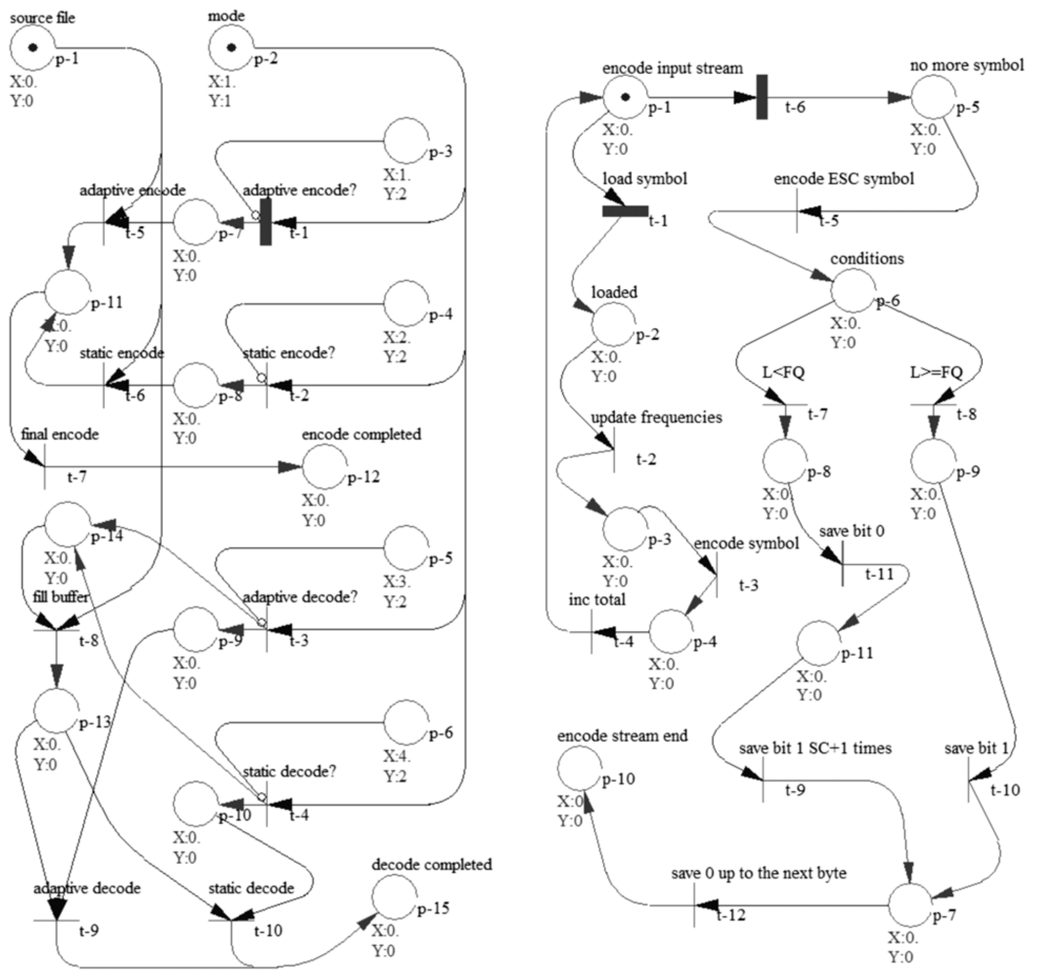


Fig. 1. A UPN model of arithmetic coding compression and decompression (left); a UPN model of adaptive arithmetic coding compression of an input stream of symbols (right).

A sequence $\{t_4\} - \{t_7\} - \{t_9\} - \{t_{11}\}$ does $e1$ scaling: saves bit 0, then recalculates values:

$$Low = 2 \cdot Low, \qquad High = 2 \cdot High + 1, \tag{4.4}$$

and then saves bit 1 $e3$ scaling counter (SC) times.

A sequence $\{t_5\} - \{t_8\} - \{t_{10}\} - \{t_{12}\}$ does $e2$ scaling: saves bit 1, then concurrently recalculates values:

$$Low = 2 \cdot (Low - Hf), \qquad High = 2 \cdot (High - Hf) + 1, \tag{4.5}$$

then saves bit 0 $e3$ scaling counter (SC) times.

After any one of the last two sequences, a set of transitions $\{t_{13}, t_{18}\}$ is enabled. At this moment only one of these transitions will be activated because of $\mu(p_{11}) = 1$ and according to the conditions 1°–3°.

Firing of transition t_{13} changes marking $\mu(p_{12}) = 1$ which represents the end of a symbol encoding. Firing of the transition t_{18} starts a new iteration of checking conditions for $e1$, $e2$ or $e3$ scaling. Firing of the transition t_4 shows that the $e1$ scaling condition is true and $e1$ scaling starts again, while firing of transition t_5 shows that the $e2$ scaling condition is true and $e2$ scaling starts again. Firing of transition t_6 means that the $e3$ scaling update condition is true and activates an $e3$ scaling update. This model shows that after the $e1$ scaling loop or $e2$ scaling loop follows checking of the $e3$ scaling update condition.

The update $e3$ scaling sequence is $\{t_6\} - \{t_{14}, t_{15}, t_{16}\}$. This sequence can do three parallel things: increment $e3$ *scaling counter* (SC), and calculate

$$Low = 2 \cdot (Low - FQ) \quad \text{and} \quad High = 2 \cdot (High - FQ) + 1. \tag{4.6}$$

After this sequence of transition firing, two transitions $\{t_{17}, t_{19}\}$ are enabled. At this moment only one of these transitions will be activated because of $\mu(p_{14}) = 3$ and according to the conditions 1°–3°. By firing of the transition t_{19} the state of the net goes to the new iteration of $e3$ scaling update.

The second case is that transition t_{17} will be activated and this means the end of the symbol encoding.

Fig. 2 (right) shows a UPN model of adaptive arithmetic coding decompression at an input stream level of representation.

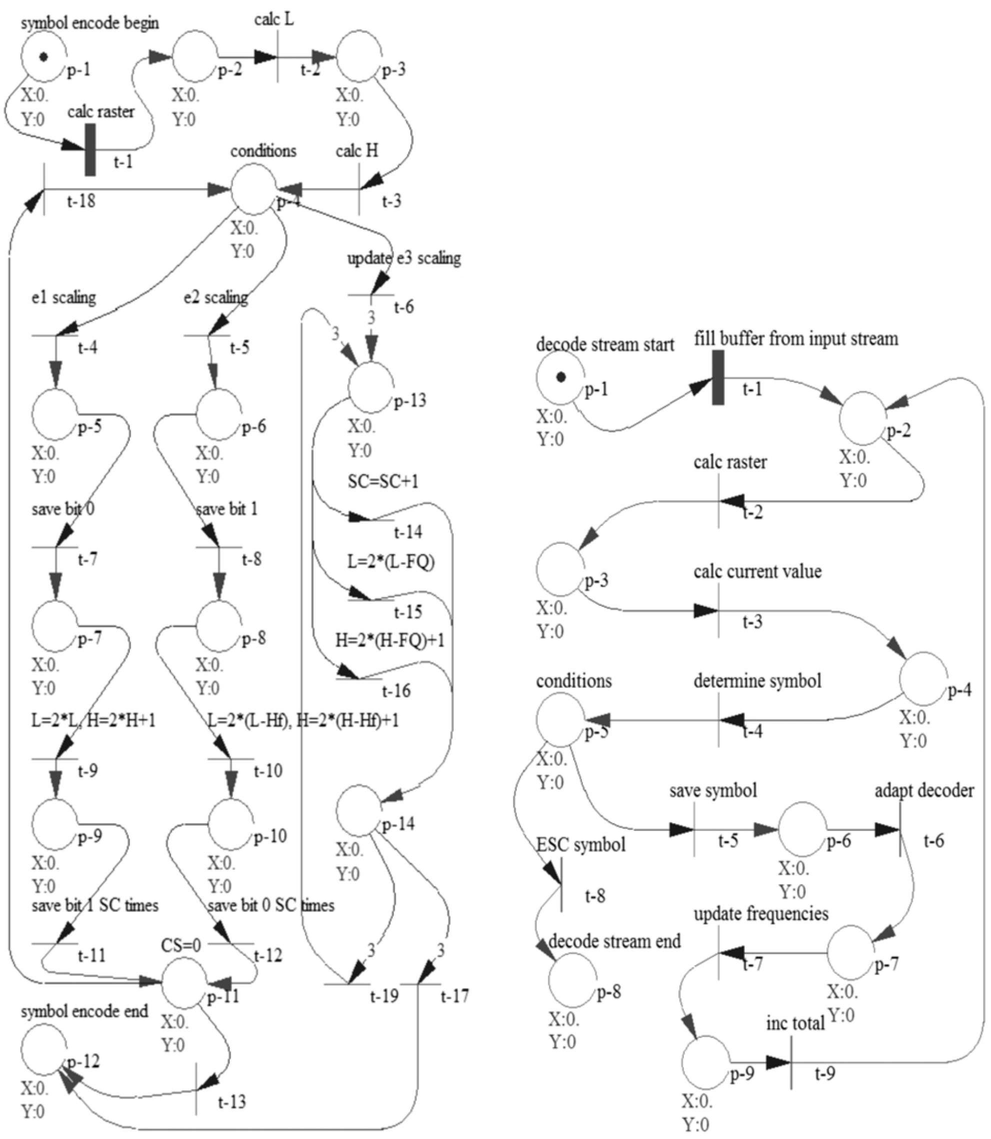


Fig. 2. A UPN model of adaptive arithmetic coding compression which refers to a symbol encode (left); a UPN model of adaptive arithmetic coding decompression of an input stream (right).

Initial marking $\mu(p_1) = 1$ determines that a sequence of transition firing $\{t_1\}-\{t_2\}-\{t_3\}-\{t_4\}$ occurs. This sequence fills the coding buffer from the input stream and calculates raster (we split the number space into a single step), after that the current value will be calculated and finally the symbol will be determined. At this moment $\mu(p_1) = 1$ and two transitions t_5 and t_8 are enabled. Only one of these transitions will be activated according to conditions 1°–3°. Now we have two cases.

The first case is a sequence of transitions firing $\{t_5\} - \{t_6\} - \{t_7\} - \{t_9\}$ which saves the symbol, than adapts the decoder, updates the frequencies, increments the total of saved symbols and finally goes to calculate the raster again. Now the sequence $\{t_2\} - \{t_3\} - \{t_4\}$ occurs as mentioned earlier, i.e., a new iteration of encoding symbol starts again.

The second case is a firing of transition t_8 which represents a decoding of the input stream. This means that the *ESC* symbol is decoded from the input stream.

Fig. 3 (left) shows a UPN model of adaptive arithmetic coding decompression which refers to an update of the decoder. Initial marking $\mu(p_1) = 1$ determines that a sequence of transition firing $\{t_1\} - \{t_2\}$ occurs. This sequence calculates values in the order of *High* then *Low* according to formulas (3.2), (3.3).

At this moment $\mu(p_3) = 1$ and this state of model checks conditions for: $e1$ scaling, $e2$ scaling and for $e3$ scaling. The condition for $e1$ scaling is that *High* is less than half of the coding interval value. The condition for $e2$ scaling is that *Low* is greater than or equal to a half of the coding interval. The condition for $e3$ scaling update is that *Low* is greater than or equal to the first quarter of the coding interval and that *High* is less than the third quarter of the interval value.

The first case is a sequence of transition firing $\{t_3\} - \{t_6\} - \{t_7\}$. This sequence refers to $e1$ scaling, and it calculates concurrently values *Low* and *High* according to formulas (4.4), then updates buffer *B*:

$$B = 2 \cdot B + LB,$$

then sets $e3$ scaling counter (SC) to zero, and finally goes into the new iteration cycle by setting $\mu(p_3) = 1$.

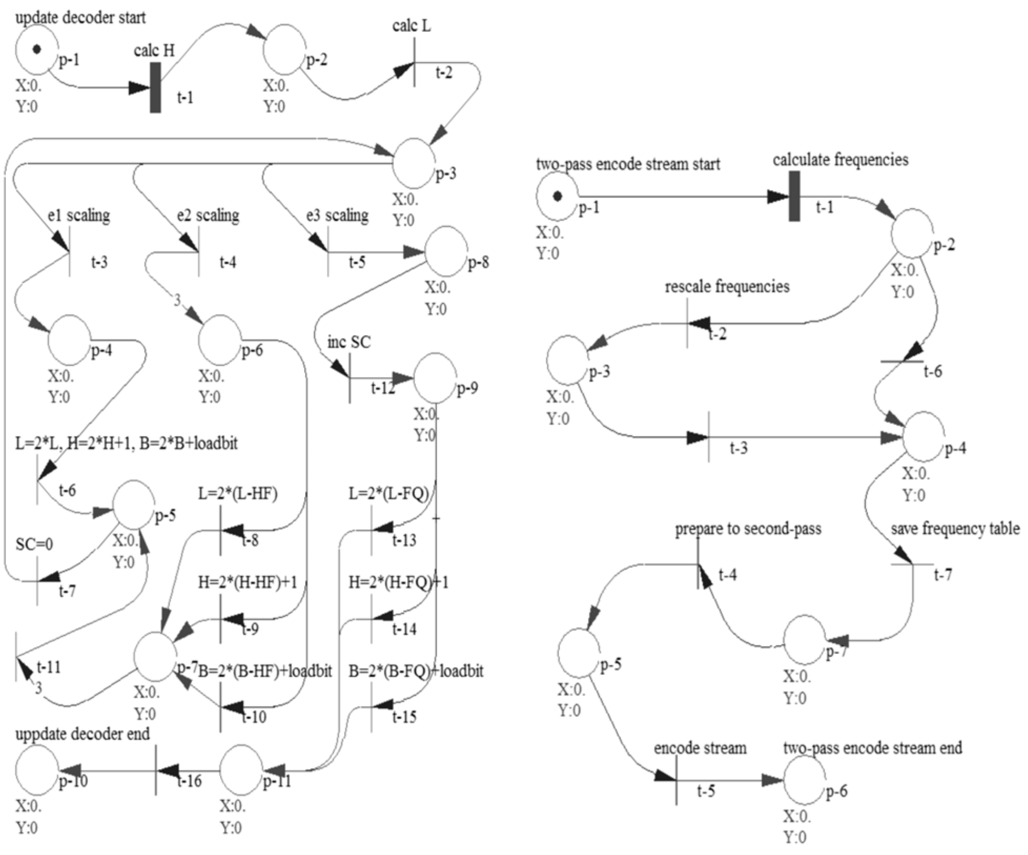


Fig. 3. A UPN model of adaptive arithmetic coding decompression which refers to an update of the decoder (left); a UPN model of two-pass static arithmetic coding compression (right).

The second case is a sequence of transition firing $\{t_4\} - \{t_8, t_9, t_{10}\} - \{t_{11}\} - \{t_7\}$. This sequence concurrently calculates three values: *Low* and *High* according to formulas (4.5) and buffer

$$B = 2 \cdot (B - Hf) + LB.$$

Before firing of the transition t_{11} marks $\mu(p_7) = 3$ and $\mu(p_5) = 0$ while the marking after firing of the transition t_{11} will become $\mu(p_7) = 0$ and $\mu(p_5) = 1$. It saves a number of tokens in the net. At last, the sequence sets $e3$ scaling counter (SC) to zero. After this, by setting $\mu(p_3) = 1$ the new iteration cycle of conditions checking starts.

The third case is a sequence of transition firing $\{t_5\} - \{t_{12}\} - \{t_{13}, t_{14}, t_{15}\} - \{t_{16}\}$. The sequence increments the counter of $e3$ scaling (SC) then concurrently calculates values *Low* and *High* according to the formulas (4.6) and buffer

$$B = 2 \cdot (B - FQ) + LB.$$

After this firing of the transition, t_{16} saves a number of tokens in this net as in the previous case and sets a new state of the model which means the end of the update decoder.

4.2. Static data model

Static arithmetic coding uses two passes or fixed statistical models and performs only one-pass over the data. In the two-pass case the first pass makes a statistical model. The second pass compresses the data. This reduces the total compression coding efficiency because this statistical model has to be transmitted to the decoder.

Fig. 3 (right) shows a UPN model of two-pass static arithmetic compression at an input stream level of representation. Initial marking $\mu(p_1) = 1$ determines that transition t_1 is enabled and will be fired. This firing calculates frequencies of the symbol over the input stream. The state of the UPN model is $\mu(p_2) = 1$ and there are two enabled transitions t_2 and t_6. Only one of these transitions will be fired according to the conditions 1°–3°.

The first case is that the transition t_2 will be fired. This means that UPN model rescales the frequencies. After this transition t_3 is enabled and will be fired and now is $\mu(p_4) = 1$.

The second case is that transition t_6 will be fired, and this firing also sets $\mu(p_4) = 1$. The second case avoids frequency rescaling.

After both of these cases a sequence of transition firing $\{t_7\} - \{t_4\} - \{t_5\}$ occurs. This sequence saves frequency table to the output stream, then prepares second pass of encoding (update pointers to the input steam), then decode input stream by using the frequency table.

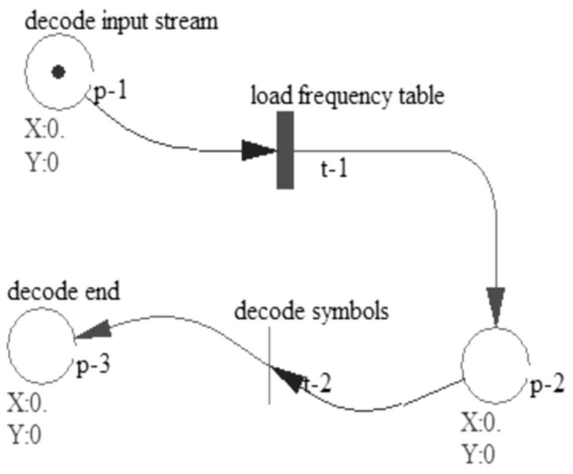


Fig. 4. An UPN model of arithmetic coding decompression of a static encoded input stream.

Fig. 4 shows a UPN model of arithmetic coding decompression of a static encoded input stream. This very simple model represents an initial marking $\mu(p_1) = 1$ which makes a sequence of transition firing $\{t_1\} - \{t_2\}$. This sequence loads the frequency table from the beginning of the input stream by using data, and then decodes the rest of the data which represent encoded symbols.

4.3. *A UPN model execution*

By executing the UPN model for given initial marking shown in Fig. 1 (left) the next sequence of transitions firing will happen $\{t_1\} - \{t_5\} - \{t_7\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}&(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\&(1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) = (\text{dead node}).\end{aligned}$$

A dead node presents a state when there are no enabled transitions in the net.

Other possible cases are determined by value of $p_2.x$ (meaning attribute x of the place p_2) as follows:

$p_2.x = 2$ determines the sequences $\{t_2\} - \{t_6\} - \{t_7\}$ and

$$\begin{aligned}&(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\&(1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) = (\text{dead node}).\end{aligned}$$

$p_2.x = 3$ determines the sequences $\{t_3\} - \{t_8\} - \{t_9\}$ and

$$\begin{aligned}&(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\&(1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1) = (\text{dead node}).\end{aligned}$$

$p_2.x = 4$ determines the sequences $\{t_4\} - \{t_8\} - \{t_{10}\}$ and

$$\begin{aligned}&(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\&(1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0),\\&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1) = (\text{dead node}).\end{aligned}$$

By executing the UPN model for a given initial marking shown in Fig. 1 (right) the next sequences of transitions firing will happen:

Sequence 2.1: $\{t_1\} - \{t_2\} - \{t_3\} - \{t_4\}$ and appropriate sequence of marking vectors as follows:

$$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),$$

$(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$,
$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (double node, init. marking).

Sequence 2.2: $\{t_6\} - \{t_5\} - \{t_7\} - \{t_{11}\} - \{t_9\} - \{t_{12}\}$ and appropriate sequence of marking vectors as follows:

$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (initial marking),
$(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$,
$(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0) =$ (dead node).

Sequence 2.3: $\{t_6\} - \{t_5\} - \{t_8\} - \{t_{10}\} - \{t_{12}\}$ and appropriate sequence of marking vectors as follows:

$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (initial marking),
$(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0) =$ (dead node).

By executing the UPN model for a given initial marking shown in Fig. 2 (left), the next sequences (mark this as 3.1) of transitions firing will happen $\{t_1\} - \{t_2\} - \{t_3\}$ and the appropriate sequence of marking vectors is:

$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (initial marking),
$(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$.

There are several sequences that will be possible from this net state as follows (mark them as 3.2.1, 3.2.2 and 3.2.3, respectively):

Sequence 3.2.1: $\{t_4\} - \{t_7\} - \{t_9\} - \{t_{11}\}$ and appropriate sequence of marking vectors as follows:

$(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (will be double node),
$(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) =$ (double node).

Sequence 3.2.2: $\{t_5\} - \{t_8\} - \{t_{10}\} - \{t_{12}\}$ and an appropriate sequence of marking vectors as follows:

$(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (will be double node),
$(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$,
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) =$ (double node).

From this double node which appears also at the end of the sequence 3.2.1, there are two possible sequences.

The first one is $\{t_{18}\}$ with its appropriate sequence of marking vectors as follows:

$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) =$ (double node),
$(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =$ (double node).

The second one is $\{t_{13}\}$ and an appropriate sequence of marking vectors as follows:

$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0) =$ (double node),
$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0) =$ (dead node).

Sequence 3.2.3 includes two cases. The first one is a sequence $\{t_6\} - \{t_{16}\} - \{t_{19}\}$ and an appropriate sequence of marking vectors as follows:

$$\begin{aligned}&(0,0,0,1,0,0,0,0,0,0,0,0,0,0) = (\text{double node}),\\&(0,0,0,0,0,0,0,0,0,0,0,0,1,0) = (\text{will be double node}),\\&(0,0,0,0,0,0,0,0,0,0,0,0,0,1),\\&(0,0,0,0,0,0,0,0,0,0,0,0,1,0) = (\text{double node}).\end{aligned}$$

These marking vectors include the cyclic sequence $\{t_{16}\} - \{t_{19}\}$.

The second sequence is a sequence $\{t_6\} - \{t_{16}\} - \{t_{17}\}$ and an appropriate sequence of marking vectors as follows:

$$\begin{aligned}&(0,0,0,1,0,0,0,0,0,0,0,0,0,0) = (\text{double node}),\\&(0,0,0,0,0,0,0,0,0,0,0,0,1,0) = (\text{double node}),\\&(0,0,0,0,0,0,0,0,0,0,0,0,0,1) = (\text{double node}),\\&(0,0,0,0,0,0,0,0,0,0,0,1,0,0) = (\text{dead node}).\end{aligned}$$

By executing the UPN model for a given initial marking shown in Fig. 2 (right), the two sequences of transitions firing will happen.

The first one $\{t_1\} - \{t_2\} - \{t_3\} - \{t_4\} - \{t_8\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}&(1,0,0,0,0,0,0,0,0) = (\text{initial marking}),\\&(0,1,0,0,0,0,0,0,0) = (\text{will be double node}),\\&(0,0,1,0,0,0,0,0,0),\\&(0,0,0,1,0,0,0,0,0),\\&(0,0,0,0,1,0,0,0,0),\\&(0,0,0,0,0,0,0,1,0) = (\text{dead node}).\end{aligned}$$

The second one $\{t_1\} - \{t_2\} - \{t_3\} - \{t_4\} - \{t_5\} - \{t_6\} - \{t_7\} - \{t_9\}$ and appropriate sequence of marking vectors is:

$$\begin{aligned}&(1,0,0,0,0,0,0,0,0) = (\text{initial marking}),\\&(0,1,0,0,0,0,0,0,0) = (\text{will be double node}),\\&(0,0,1,0,0,0,0,0,0),\\&(0,0,0,1,0,0,0,0,0),\\&(0,0,0,0,1,0,0,0,0),\\&(0,0,0,0,0,1,0,0,0),\\&(0,0,0,0,0,0,1,0,0),\\&(0,0,0,0,0,0,0,0,1),\\&(0,1,0,0,0,0,0,0,0) = (\text{double node}).\end{aligned}$$

By executing the UPN model for a given initial marking shown in Fig. 3 (left) the sequence of transitions firing $\{t_1\} - \{t_2\}$ will happen and an appropriate sequence of marking vectors is:

$$\begin{aligned}&(1,0,0,0,0,0,0,0,0,0,0) = (\text{initial marking}),\\&(0,1,0,0,0,0,0,0,0,0,0),\\&(0,0,1,0,0,0,0,0,0,0,0) = (\text{will be double node}).\end{aligned}$$

There are three sequences possible from this state. The first one is $\{t_3\} - \{t_6\} - \{t_7\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}&(0,0,1,0,0,0,0,0,0,0,0) = (\text{will be double node}),\\&(0,0,0,1,0,0,0,0,0,0,0),\\&(0,0,0,0,1,0,0,0,0,0,0) = (\text{will be double node}),\\&(0,0,1,0,0,0,0,0,0,0,0) = (\text{double node}).\end{aligned}$$

The second sequence is $\{t_4\} - \{t_8, t_9, t_{10}\} - \{t_{11}\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}&(0,0,1,0,0,0,0,0,0,0,0) = (\text{double node}),\\&(0,0,0,0,0,1,0,0,0,0,0),\\&(0,0,0,0,0,0,1,0,0,0,0),\\&(0,0,0,0,1,0,0,0,0,0,0) = (\text{double node}).\end{aligned}$$

The third sequence is $\{t_5\} - \{t_{12}\} - \{t_{13}, t_{14}, t_{15}\} - \{t_{16}\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}
&(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0) = (\text{double node}),\\
&(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0),\\
&(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0),\\
&(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1) = (\text{dead node}).
\end{aligned}$$

By executing the UPN model for given initial marking shown Fig. 3 (right) the two sequences of transitions firing will happen. The first sequence is $\{t_1\} - \{t_2\} - \{t_3\} - \{t_7\} - \{t_4\} - \{t_5\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}
&(1, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\
&(0, 1, 0, 0, 0, 0, 0),\\
&(0, 0, 1, 0, 0, 0, 0),\\
&(0, 0, 0, 1, 0, 0, 0),\\
&(0, 0, 0, 0, 0, 0, 1),\\
&(0, 0, 0, 0, 1, 0, 0),\\
&(0, 0, 0, 0, 0, 1, 0) = (\text{dead node}).
\end{aligned}$$

The second sequence is $\{t_1\} - \{t_6\} - \{t_7\} - \{t_4\} - \{t_5\}$ and an appropriate sequence of marking vectors is:

$$\begin{aligned}
&(1, 0, 0, 0, 0, 0, 0) = (\text{initial marking}),\\
&(0, 1, 0, 0, 0, 0, 0),\\
&(0, 0, 0, 1, 0, 0, 0),\\
&(0, 0, 0, 0, 0, 0, 1),\\
&(0, 0, 0, 0, 1, 0, 0),\\
&(0, 0, 0, 0, 0, 1, 0) = (\text{dead node}).
\end{aligned}$$

By executing the UPN model for a given initial marking shown in Fig. 4 a trivial sequence of transitions firing will happen: $\{t_1\} - \{t_2\}$ and an appropriate sequence of marking vectors is: $(1, 0, 0, 0, 0, 0, 0) = (\text{initial marking})$; $(0, 1, 0, 0, 0, 0, 0)$; $(0, 0, 1, 0, 0, 0, 0) = (\text{dead node})$.

5. Experimental results

According to all UPN models shown in this paper, an application is created. The application compresses and decompresses several sets of test files [20] by using arithmetic coding with respect to the input parameters. These input parameters are: symbol type (byte or nibble), adaptive or static data model and frequency scaling (scaled or non-scaled). If we use the frequency scaled option there are two cases: frequency table divided by 2, and proportional scaling frequency table from min frequency equals 1 to max frequency equals 255. In all possible cases of our arithmetic coding, the initial (minimum) frequency for all symbols is 1.

In this section we use some abbreviations for various types of arithmetic coding which depend on these input parameters. These abbreviations are as follows:

- *Type*_10 represents that an input symbol is a byte, we use adaptive arithmetic coding with non-scaled frequency table;
- *Type*_11 represents that an input symbol is a byte, we use adaptive arithmetic coding with divby2-scaled frequency table;
- *Type*_12 represents that an input symbol is a byte, we use adaptive arithmetic coding with max255-scaled frequency table;
- *Type*_20 represents that an input symbol is a byte, we use static arithmetic coding with non-scaled frequency table;
- *Type*_21 represents that an input symbol is a byte, we use static arithmetic coding with divby2-scaled frequency table;
- *Type*_22 represents that an input symbol is a byte, we use static arithmetic coding with max255-scaled frequency table;
- *Type*_30 represents that an input symbol is a nibble, we use adaptive arithmetic coding with non-scaled frequency table;
- *Type*_31 represents that an input symbol is a nibble, we use adaptive arithmetic coding with divby2-scaled frequency table;
- *Type*_32 represents that an input symbol is a nibble, we use adaptive arithmetic coding with max255-scaled frequency table;
- *Type*_40 represents that an input symbol is a nibble, we use static arithmetic coding with non-scaled frequency table;
- *Type*_41 represents that an input symbol is a nibble, we use static arithmetic coding with divby2-scaled frequency table;
- *Type*_42 represents that an input symbol is a nibble, we use static arithmetic coding with max255-scaled frequency table;

Table 5.1
File sizes (bytes) in arithmetic coding compression *Type*_10, *Type*_11, *Type*_12; *Type*_20, *Type*_21, *Type*_22 over texture files.

Name of `tiff` file & its size		*Type*_10	*Type*_11	*Type*_12	*Type*_20	*Type*_21	*Type*_22
`texmos1.p512`	262,278	259,557	259,591	259,597	260,418	260,415	260,414
`texmos2.p512`	262,278	254,538	254,584	254,589	255,394	255,392	255,404
`texmos2.s512`	262,278	98,886	99,177	105,951	99,602	99,641	105,227
`texmos3.p512`	262,278	255,917	255,952	255,950	256,775	256,772	256,777
`texmos3.s512`	262,278	98,358	98,644	108,202	99,073	99,113	105,874
`texmos3b.p512`	262,278	255,924	255,960	255,958	256,782	256,779	256,784
`1.3.01`	1,048,710	975,783	975,809	975,924	976,626	976,582	976,810
`1.3.02`	1,048,710	965,618	965,666	965,958	966,459	966,416	966,699
`1.3.03`	1,048,710	956,633	956,712	957,079	957,466	957,425	957,751
`1.3.04`	1,048,710	953,032	953,057	953,285	953,882	953,836	953,935
`1.3.05`	1,048,710	999,535	999,545	999,533	1,000,392	1,000,346	1,000,354

Table 5.2
File sizes (bytes) in arithmetic coding compression *Type*_30, *Type*_31, *Type*_32; *Type*_40, *Type*_41, *Type*_42 over texture files.

Name of `tiff` file & its size		*Type*_30	*Type*_31	*Type*_32	*Type*_40	*Type*_41	*Type*_42
`texmos1.p512`	262,278	262,259	262,254	262,272	262,318	262,307	262,319
`texmos2.p512`	262,278	262,144	262,139	262,158	262,203	262,192	262,205
`texmos2.s512`	262,278	173,773	173,787	175,391	173,827	173,815	175,059
`texmos3.p512`	262,278	262,153	262,150	262,169	262,213	262,202	262,214
`texmos3.s512`	262,278	98,358	98,644	108,202	172,782	172,771	174,038
`texmos3b.p512`	262,278	262,156	262,152	262,171	262,215	262,203	262,216
`1.3.01`	1,048,710	1,020,475	1,020,385	1,020,425	1,020,705	1,020,520	1,020,485
`1.3.02`	1,048,710	1,014,635	1,014,546	1,014,652	1,014,869	1,014,682	1,014,667
`1.3.03`	1,048,710	1,012,663	1,012,572	1,012,669	1,012,891	1,012,708	1,012,684
`1.3.04`	1,048,710	1,007,214	1,007,125	1,007,247	1,007,445	1,007,260	1,007,259
`1.3.05`	1,048,710	1,030,220	1,030,130	1,030,163	1,030,443	1,030,263	1,030,209

Table 5.3
File sizes (bytes) in arithmetic coding compression *Type*_10, *Type*_11, *Type*_12; *Type*_20, *Type*_21, *Type*_22 over aerial files.

Name of file & its size		*Type*_10	*Type*_11	*Type*_12	*Type*_20	*Type*_21	*Type*_22
`2.1.02.tiff`	786,572	721,177	721,273	721,515	722,008	721,986	722,113
`2.1.08.tiff`	786,572	628,095	628,248	629,552	628,878	628,866	630,392
`2.1.12.tiff`	786,572	646,185	646,309	646,309	646,972	646,960	648,214
`2.2.03.tiff`	3,145,868	2,490,157	2,490,183	2,499,028	2,491,287	2,490,883	2,497,750
`2.2.13.tiff`	3,145,868	2,892,415	2,892,309	2,892,888	2,893,595	2,893,186	2,893,682
`2.2.17.tiff`	3,145,868	2,799,801	2,799,744	2,801,846	2,800,959	2,800,548	2,802,071
`2.2.24.tiff`	3,145,868	2,884,736	2,884,641	2,885,452	2,885,915	2,885,502	2,886,077
`3.2.25.tiff`	1,048,710	883,060	883,204	884,926	883,866	883,830	885,451

The main goal in this experiment is to verify UPN models transformed into the real application and to explore our 12 types of arithmetic coding compression over the used test files [20].

The first two tables (Tables 5.1 and 5.2) show experimental results which refer to the arithmetic coding compression of some test texture files [20]. The compressed files shown in Table 5.1 (second part) include an appropriate frequency table which is 1024 bytes long because of the byte input symbol. The compressed files shown in Table 5.2 (second part) include an appropriate frequency table which is 64 bytes long because of the nibble input symbol.

The next two tables (Tables 5.3 and 5.4) show experimental results refer to arithmetic coding compression of some test aerial files [20]. The compressed files shown in Table 5.3 (second part) include an appropriate frequency table which is 1024 bytes long (byte input symbol). The compressed files shown in Table 5.4 (second part) include an appropriate frequency table which is 64 bytes long (nibble input symbol).

Finally, the last two tables (Tables 5.5 and 5.6) show experimental results which refer to the arithmetic coding compression of some test misc files [20]. The compressed files shown in Table 5.5 (second part) include an appropriate frequency table which is 1024 bytes long (byte input symbol). The compressed files shown in Table 5.6 (second part) include an appropriate frequency table which is 64 bytes long (nibble input symbol).

According to the results of arithmetic compression shown in the tables, for our set of test files we can choose *Type*_10 (an input symbol is a byte, we use an adaptive model with a non-scaled frequency table) as a suitable method in our set of 12 types of arithmetic compression.

At last we decompressed all our compressed files to test our application and the UPN models which refer to arithmetic decompression.

Table 5.4
File sizes (bytes) in arithmetic coding compression *Type*_30, *Type*_31, *Type*_32; *Type*_40, *Type*_41, *Type*_42 over aerial files.

Name of file & its size		*Type*_30	*Type*_31	*Type*_32	*Type*_40	*Type*_41	*Type*_42
`2.1.02.tiff`	786,572	759,655	759,606	759,707	759,805	759,701	759,716
`2.1.08.tiff`	786,572	714,532	714,482	714,672	714,681	714,578	714,730
`2.1.12.tiff`	786,572	725,919	725,870	726,020	726,071	725,965	726,084
`2.2.03.tiff`	3,145,868	2,860,031	2,859,201	2,859,593	2,861,763	2,860,071	2,859,441
`2.2.13.tiff`	3,145,868	3,047,532	3,046,704	3,046,353	3,049,369	3,047,596	3,046,421
`2.2.17.tiff`	3,145,868	3,009,587	3,008,758	3,008,537	3,011,281	3,009,626	3,008,494
`2.2.24.tiff`	3,145,868	3,046,918	3,046,087	3,045,762	3,048,612	3,046,962	3,045,805
`3.2.25.tiff`	1,048,710	979,211	979,121	979,305	979,442	979,256	979,329

Table 5.5
File sizes (bytes) in arithmetic coding compression *Type*_10, *Type*_11, *Type*_12; *Type*_20, *Type*_21, *Type*_22 over misc files.

Name of file & its size		*Type*_10	*Type*_11	*Type*_12	*Type*_20	*Type*_21	*Type*_22
`4.2.03.tiff`	786,572	763,514	763,561	763,584	764,368	764,342	764,323
`5.1.09.tiff`	65,670	55,305	55,383	55,335	56,165	56,166	56,175
`5.1.14.tiff`	65,670	60,504	60,563	60,524	61,377	61,378	61,379
`5.3.02.tiff`	1,048,710	895,816	895,895	896,269	896,646	896,603	897,291
`7.1.07.tiff`	262,278	196,833	197,042	197,536	197,615	197,632	198,282
`boat.512.tiff`	262,278	236,053	236,191	236,425	236,898	236,895	236,936

Table 5.6
File sizes (bytes) in arithmetic coding compression *Type*_30, *Type*_31, *Type*_32; *Type*_40, *Type*_41, *Type*_42 over misc files.

Name of file & its size		*Type*_30	*Type*_31	*Type*_32	*Type*_40	*Type*_41	*Type*_42
`4.2.03.tiff`	786,572	777,736	777,686	777,740	777,886	777,783	777,770
`5.1.09.tiff`	65,670	60,748	60,750	60,772	60,799	60,799	60,813
`5.1.14.tiff`	65,670	63,514	63,515	63,525	63,565	63,564	63,572
`5.3.02.tiff`	1,048,710	980,077	97,998	980,122	980,309	980,122	980,218
`7.1.07.tiff`	262,278	239,514	239,511	239,593	239,573	239,562	239,615
`boat.512.tiff`	262,278	248,767	248,764	248,879	248,827	248,815	248,867

6. Conclusion

Upgraded Petri nets are suitable for modeling various types of arithmetic coding at any level. Original software for modeling and simulations of an Upgraded Petri net, PeM (Petri net Manager), is developed and used for all models described in this paper. Several UPN models are shown. These models represent arithmetic coding from a general level of representation up to a register level of representation. By executing given UPN models of an arithmetic coding, the models pass through many states which are the states of the arithmetic coding algorithm. After careful analysis of the models we transform these models into the real application. By using this application and PeM we do several cycles of modeling, simulation and analysis and finally check the suitability of given UPN models. After that we use our twelve types of arithmetic coding over the set of test images. These types are determined by input parameters: symbol type, adaptive or static data set and scale frequency table type. Finally, we represent results of all of these types of arithmetic coding over the sets of some test images (textures, aerial, misc). All models are observed in the experiments by using a created application over a standard set of files. Experimental results are presented and compared.

Acknowledgment

This research was in part supported by the Serbian Ministry of Education, Science and Technological Development.

References

[1] P.S. Štrbac, An approach to modeling communication protocol by using upgraded Petri nets, Ph.D. Dissertation, Military Academy, Belgrade, Serbia, 2002.
[2] T. Murata, Petri nets: properties, analysis and applications, Proc. IEEE 77 (4) (1989) 541–580.
[3] D. Gašević, V. Devedžić, Teaching Petri nets using P3, IEEE Educ. Tech. Soc. 7 (4) (2004) 153–166.
[4] I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic coding for data compression, Commun. ACM 30 (1987) 520–540.
[5] A. Moffat, R.M. Neal, I.H. Witten, Arithmetic coding revisited, ACM Trans. Inf. Syst. 16 (1998) 256–294.
[6] M. Grangetto, E. Magli, G. Olmo, Distributed arithmetic coding, IEEE Commun. Lett. 11 (11) (2007) 883–885.
[7] M. Grangetto, E. Magli, G. Olmo, Rate-compatible distributed arithmetic coding, IEEE Commun. Lett. 12 (8) (2008) 575–577.
[8] D.J.C. MacKay, Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003.
[9] P.G. Howard, J.S. Witter, Arithmetic coding for data compression, Proc. IEEE 82 (6) (1994) 857–865.
[10] A. Said, Introduction to Arithmetic Coding — Theory and Practice, Imaging Systems Laboratory HP Laboratories, Palo Alto, 2004, HPL-2004-76.
[11] G.G. Langdon Jr., An introduction to arithmetic coding, IBM J. Res. Dev. 28 (2) (1984) 135–149.

[12] H. Kim, J. Wen, J.D. Villasenor, Secure arithmetic coding, IEEE Trans. Signal Process. 55 (5) (2007) 2263–2272.
[13] B. Zhu, E. Yang, A.H. Tewfik, Arithmetic coding with dual symbol sets and its performance analysis, IEEE Trans. Image Process. 8 (12) (1999) 1667–1676.
[14] Rung-Ching Chen, Pei-Yan Pai, Yung-Kuan Chan, Chin-Chen Chang, Lossless image compression based on multiple-tables arithmetic coding, Math. Probl. Eng. 2009 (2009) 13. http://dx.doi.org/10.1155/2009/128317. Article ID 128317.
[15] R. Pasco, Source coding algorithms for fast data compression, Stanford University, Ph.D. Thesis, 1976.
[16] J.J. Rissanen, Generalized kraft inequality and arithmetic coding, IBM J. Res. Dev. (1976) 198–203.
[17] F. Rubin, Arithmetic stream coding using fixed precision registers, IEEE Trans. Inf. Theory IT-25 (6) (1979) 672–675.
[18] J.J. Rissanen, G.G. Langdon, Arithmetic coding, IBM J. Res. Dev. (1979) 146–162.
[19] M. Guazzo, A general minimum-redundancy source-coding algorithm, IEEE Trans. Inf. Theory IT-26 (1) (1980) 15–25.
[20] The USC-SIPI image database, University of Southern California: http://sipi.usc.edu/database/.