

УДК 004.432.4

10.25209/2079-3316-2023-14-4-91-122



## Цветные сети Петри и язык распределенного программирования UPL: их сравнение и перевод

Аркадий Валентинович Климов<sup>✉</sup>

Институт проблем проектирования в микроэлектронике РАН

<sup>✉</sup>arkady.klimov@gmail.com

**Аннотация.** Сети Петри широко используются как средство моделирования распределенных мультиагентных систем. Существуют инструменты работы с расширенными сетями Петри, в которых токены нагружены произвольными данными. В частности, CPN Tools позволяет описывать, проигрывать и исследовать цветные сети Петри (Coloured Petri Nets, CPN). Ставится вопрос о возможности использовать этот инструмент для разработки, прототипирования и исследования параллельных распределенных вычислительных алгоритмов, в идеале — превращения их в работающие эффективные параллельные программы. У нас есть опыт экспериментального программирования разных алгоритмов в нашем графическом языке UPL, который пока существует как бы «на бумаге». Его сравнение с CPN показывает, что в их семантиках много общего. В статье оба языка определяются, сравниваются на примерах и через правила перевода из одного в другой. Также описываются средства управления распределением вычислений для UPL. Интересен вопрос об их переносе в CPN, где им пока аналога нет.

**Ключевые слова и фразы:** сети Петри, цветные сети Петри, параллельное программирование, потоковая модель вычислений, граф алгоритма, графическое программирование, язык UPL, функция распределения

**Благодарности:** Работа поддержана ИППМ РАН

**Для цитирования:** Климов Ар. В. *Цветные сети Петри и язык распределенного программирования UPL: их сравнение и перевод* // Программные системы: теория и приложения. 2023. Т. 14. № 4(59). С. 91–122. [https://psta.pstiras.ru/read/psta2023\\_4\\_91-122.pdf](https://psta.pstiras.ru/read/psta2023_4_91-122.pdf)

## Введение

Сети Петри [1, 2] широко используются как средство моделирования распределенных мультиагентных систем самых разных видов: от технических установок [3] до больших интегральных схем [4] и нейронных сетей [5]. Существуют инструменты работы с расширенными сетями Петри, в которых токены нагружены произвольными данными. В частности, *CPN Tools*<sup>UR</sup> позволяет описывать, проигрывать и исследовать цветные сети Петри (Coloured Petri Nets, CPN)[6].

В Цветных Сетях Петри (ЦСП) каждый переход может сопровождаться вычислением новых выходных значений из старых входных. Это приводит к возможности описания на языке CPN не только моделей распределенных систем, но вычислительных алгоритмов широкого класса.

У нас накоплен большой опыт экспериментального программирования в языке потоков данных UPL [7, 8]. Процесс выполнения CPN во многом похож на вычислительный процесс в языке UPL. В последнем так же, как в CPN переходы, имеют узлы-шаблоны, определяющие правила локальных трансформаций множеств токенов. Это множество, как и разметка в CPN, представляет собой состояние вычислительного процесса. Процесс смены состояний состоит из срабатываний узлов, как и переходов в CPN, образуя виртуальный вычислительный граф, переводящий состояние с исходными данными в состояние с результатом вычислений. И как в CPN, для срабатывания узла в текущем состоянии должны найтись несколько токенов с согласованными компонентами.

Название «UPL» означает универсальный язык параллельного программирования. Мы полагаем, что в нем адекватным образом может быть выражен любой вид алгоритмического параллелизма, имеющийся в различных существующих языках параллельного программирования. И хотя Цветные Сети Петри обычно не позиционируются как «язык программирования», они порождают свой особый алгоритмический параллелизм, который хотелось бы проверить на предмет его выразимости в языке UPL и наоборот.

При попытке перевода из CPN в UPL возникают трудности, показывающие недостаточную универсальность и выразительность UPL. Мы отмечаем эти трудности и задаем ограничения на CPN, при которых они не возникают. С другой стороны, эти трудности указывают на желательность введения некоторых расширений UPL для их преодоления. Рассмотрению этих расширений будет посвящена другая статья.

Перевод из UPL в CPN также в некоторых случаях, причем весьма существенных, наталкивается на трудности, показывающие недостаток

определенных возможностей CPN. Их подробное обсуждение также войдет в будущую статью.

Статья написана по следующему плану. В разделе 1 даются формальные определения сетей Петри – сначала стандартных (с «пустыми фишками»), потом цветных (где токены несут данные – «цвета»). Для обоих вариантов описывается точная семантика, причем для цветных двумя способами: непосредственно, как это принято в литературе, и путем сведения к стандартной сети Петри, возможно бесконечной. Завершается данный раздел работающим примером параллельного алгоритма сортировки. В разделе 2 язык UPL вводится сначала как результат структурных преобразований подкласса сетей CPN с указанием необходимых ограничений на них. Затем он же определяется независимо со своими правилами выполнения. В разделе 3 показаны примеры типовых фрагментов одновременно на CPN и UPL, а в разделе 4 дан более развернутый пример алгоритма на обоих языках. Сходства и особенности двух языков подробно обсуждаются в разделе 5, показывая, что в каждом имеются элементы, не выразимые в другом. В разделе 6 кратко описаны средства управления распределенным выполнением программ UPL, которые едва ли могут быть применены к CPN (оставляем это как задачу дальнейших исследований). Сравнение с близкой работой приводится в разделе 7. В заключительном разделе 9 сведены основные тезисы статьи.

## 1. Язык CPN и его семантика

Здесь дадим не совсем полное, но достаточное для наших целей полуформальное описание. Сначала напомним определение стандартных сетей Петри, а затем дадим его расширение до цветных сетей Петри.

### 1.1. Стандартные сети Петри

#### 1.1.1. Структура

(Стандартная) сеть Петри (SPN) определяется как двудольный ориентированный граф. Вершины (*nodes*) графа называются *переходами* (*transitions*)  $t \in T$  и *местами* (*places*)  $p \in P$ , а связи – *дугами*  $a \in A$ . (Множества  $T$ ,  $P$  и  $A$  попарно не пересекаются). Еще задана функция вершин  $N : A \rightarrow (P \times T) \cup (T \times P)$ , которая каждой дуге приписывает элемент либо из  $P \times T$  (входные дуги), либо из  $T \times P$  (выходные дуги). Множество входных (выходных) дуг перехода  $t$  обозначается  $\bullet t$  (соответственно,  $t \bullet$ ). Такое определение допускает наличие любого числа дуг из места  $p$  в переход  $t$  (а также из перехода  $t$  в место  $p$ ) при том, что формально это различные элементы множества дуг  $A$ .

Текущее состояние стандартной сети Петри, называемое *разметкой* (*marking*), задается как некоторое мультимножество<sup>1</sup> мест,  $M \in P_{MS}$ . Другими словами, разметка задает для каждого места  $p$  некоторое количество  $M(p)$  *токенов* (*tokens*), находящихся в этом месте. Обозначения  $\bullet t$  и  $t\bullet$  будем также толковать как мультимножества мест, соответственно, входных или выходных для перехода  $t$ .

Для сети Петри должна быть задана начальная разметка  $M_0$ . Таким образом, Стандартная Сеть Петри с разметкой задается как набор из пяти компонентов:  $SPN = (P, T, A, N, M_0)$ .

### 1.1.2. Семантика

Теперь определим поведение SPN. Будем говорить, что разметка  $M$  *допускает* (*enables*) переход  $t$ , если  $\bullet t \leq M$ . В этом случае возможно *срабатывание* (*occurrence*) перехода  $t$ , которое приводит к новой разметке  $M' = M - (\bullet t) + (t\bullet)$ . В общем случае шаг (*step*) есть непустое конечное мультимножество переходов  $Y \in T_{MS}$ . Шаг  $Y$  *допустим* (при разметке  $M$ ), если  $\sum_{t \in Y} (\bullet t) \leq M$  (суммирование проводится с учетом кратностей элементов  $Y$ ). В этом случае возможно *одновременное срабатывание* всех переходов шага  $Y$ , и оно приводит к разметке  $M' = M - \sum_{t \in Y} (\bullet t) + \sum_{t \in Y} (t\bullet)$ . Тогда говорят, что разметка  $M'$  *непосредственно достижима* из разметки  $M$ , и записывают это как  $M[Y]M'$ . Конечную последовательность шагов и разметок  $M_1[Y_1]M_2[Y_2] \dots M_n[Y_n]M_{n+1}$ , в которой  $M_i[Y_i]M_{i+1}$  для всех  $i \in \{1, 2, \dots, n\}$ , будем называть *конечной последовательностью срабатываний* с начальной разметкой  $M_1$ , конечной разметкой  $M_{n+1}$  и длиной  $n \geq 0$ . В этом случае говорят, что разметка  $M_{n+1}$  *достижима из разметки*  $M_1$ . (Аналогично можно определить бесконечную последовательность срабатываний). Множество разметок, достижимых из разметки  $M_1$ , обозначают как  $[M_1]$ . Разметка *достижима*, если она входит в  $[M_0]$ .

---

<sup>1</sup>Мультимножество (*multiset*) типа  $X$  есть функция  $h : X \rightarrow \mathbb{N}$ , которая каждому элементу  $X$  ставит в соответствие его вес – неотрицательное целое число.  $|h| = \sum_{x \in X} h(x)$  – размер мультимножества  $h$ . Обычно используют конечные мультимножества,  $|h| < \infty$ , даже если тип  $X$  бесконечный. Сложение (вычитание), умножение на число и отношение вложения мультимножеств определяют как расширение операций  $+$  ( $-$ ),  $\cdot$  и  $\leq$  в  $\mathbb{N}$  (последней в смысле *forall*). Множество (тип) всех (конечных) мультимножеств над  $X$  обозначают  $X_{MS}$ . Его элементы записывают как формальные суммы простых мультимножеств вида  $k'x$ , где  $k'$  – кратность (вес) элемента  $x$ . Пустое мультимножество обозначают  $\emptyset_X$ , или просто  $\emptyset$ .

## 1.2. Цветные сети Петри

### 1.2.1. Структура

Теперь расширим стандартные сети Петри до цветных (Coloured) сетей Петри (CPN)[6]. Понадобится некоторый язык, в котором можно определять типы данных и записывать выражения над значениями этих типов. Авторы CPN используют слегка модифицированный функциональный язык Standard ML[9] (SML), называя его CPN ML. Типы в нем называются наборами цветов (colour sets) и включают целые (int), логические (bool), интервалы целых, конечные перечисления, а также структурные типы – кортежи (products), записи (records) и списки (lists) ограниченной или неограниченной длины.

Чтобы перейти от стандартной сети Петри к цветной, надо вначале задать совокупность типов данных (наборов цветов)  $\Sigma$  и функцию цветности  $C: P \rightarrow \Sigma$ , которая каждому месту  $p$  приписывает некоторый тип  $C(p)$ . Тип места указывает тип значений, которые может содержать токен, находящийся в этом месте. А какие именно токены и в каком количестве находятся в каждом месте задает разметка. То есть, разметка  $M$  это функция, которая с каждым местом связывает мультимножество  $M(p) \in C(p)_{MS}$ . Далее, каждой дуге  $a \in A$  припишем выражение  $E(a)$  типа  $C(p)_{MS}$ , где  $p$  – место, связанное с дугой  $a$ . (Можно использовать также выражение типа  $C(p)$ , тогда оно автоматически приводится к типу  $C(p)_{MS}$  путем приписывания префикса  $1'$ ). Для каждого перехода  $t$  зададим выражение  $G(t)$  типа bool – охрану (по умолчанию – True).

Все выражения могут содержать в качестве свободных переменных только переменные из заранее заданного списка переменных  $V$ , причем для каждой переменной  $v \in V$  должен быть зафиксирован ее тип  $Type(v) \in \Sigma$ . Если задано *связывание* (binding)  $b: V \rightarrow \bigcup \Sigma$  всех переменных  $v \in V$  со значениями  $b(v)$  нужного типа  $Type(v)$ , то всякое выражение  $E(a)$  или  $G(t)$  может быть вычислено до значения типа  $C(p)_{MS}$  или bool, соответственно. Для результата вычисления будем использовать обозначение  $E(a)\langle b \rangle$ , соответственно  $G(t)\langle b \rangle$ . При этом связывание  $b$  должно содержать как минимум все (свободные) переменные данного выражения. Наконец, в качестве начальной разметки  $M_0$  следует задать для каждого места  $p$  замкнутое (не содержащее свободных переменных) выражение  $I(p)$ , которое вычисляется до некоторого конечного мультимножества типа  $C(p)_{MS}$ . Таким образом, цветная сеть Петри задается как набор из десяти компонентов:  $CPN = (P, T, A, N, \Sigma, V, C, E, G, I)$ .

### 1.2.2. Семантика

Теперь определим поведение CPN. Сделаем это двумя различными способами, и затем покажем их эквивалентность.

Первый способ — непосредственный

Этот способ используется в статьях и руководствах по CPN [6]. Нам понадобятся следующие понятия и обозначения:

**Разметка** это функция  $M$ , которая каждому месту  $p \in P$  приписывает мультимножество токенов:  $M(p) \in C(p)_{MS}$ .

**Начальная разметка**  $M_0$  задается как результат вычисления замкнутых выражений  $I(p)$ .

**Внешний токен** (token element) — это пара  $(p, v)$ , где  $v \in C(p)$ . Множество всех возможных внешних токенов данной CPN обозначим  $TE$ .

**Множество переменных перехода**  $t$ , обозначаемое  $Var(t)$  состоит из всех свободных переменных всех выражений, стоящих на дугах, связанных с  $t$ , а также в  $G(t)$ .

**Связывание** перехода  $t$  это функция  $b$ , которая каждой переменной  $v \in Var(t)$  приписывает значение  $b(v)$  типа  $Type(v)$  так, что выполняется охрана  $G(t)\langle b \rangle$ . Множество всех связываний перехода  $t$  обозначается  $B(t)$ .

**Внешнее связывание** (binding element) это пара  $(t, b)$ , где  $t \in T$  и  $b \in B(t)$ . Множество всех внешних связываний перехода  $t$  определим как  $BE(t) = \{(t, b) | b \in B(t)\}$ . Множество всех внешних связываний данной CPN обозначим  $BE$ .

**Шаг**  $Y \in BE_{MS}$  это непустое конечное мультимножество внешних связываний. Далее прилагательное «внешнее» (оно лишь как бы уточняет связывание для «внешнего» наблюдателя) будем опускать, когда из контекста понятно, о чем речь.

Определим  $A(p, t)$  как множество всех дуг из  $p$  в  $t$ .

$$A(p, t) = \{a | N(a) = (p, t)\}$$

. Аналогично  $A(t, p)$ .

Определим  $E(p, t)$  как формальную сумму всех выражений на дугах из  $A(p, t)$ . Это допустимо, поскольку типы значение всех этих выражений одинаковые. Если нет дуг из  $p$  в  $t$ , то  $E(p, t) = \emptyset$ . Аналогично  $E(t, p)$ .

Связывание  $(t, b)$  **допускается** разметкой  $M_1$ , если для всякого места  $p$  выполняется  $E(p, t)\langle b \rangle \leq M_1(p)$ . Тогда переход  $t$  (со связыванием  $b$ ) может *сработать*, породив новую разметку  $M_2 = M_1 - \sum_{p \in P} E(p, t)\langle b \rangle + \sum_{p \in P} E(t, p)\langle b \rangle$ . Иначе говоря, значения выражений  $E(p, t)\langle b \rangle$  на входных дугах показывают мультимножества токенов, которые необходимы

для срабатывания перехода  $t$  со связыванием  $b$  и которые при этом срабатывании будут из разметки удалены, а взамен будет добавлено мультимножество токенов, заданное выражением  $E(t, p)\langle b \rangle$ .

Шаг  $Y$  **допускается** разметкой  $M_1$ , если для всякого места  $p$  справедливо  $\sum_{(t,b) \in Y} E(p, t) \leq M_1(p)$ , где суммирование производится с учетом кратностей элементов  $Y$ . Тогда могут *сработать одновременно* все элементы шага  $Y$ , порождая новую разметку  $M_2$ , такую что для всякого места  $p$  выполняется  $M_2(p) = M_1(p) - \sum_{(t,b) \in Y} E(p, t) + \sum_{(t,b) \in Y} E(t, p)$ . И тогда говорят, что разметка  $M_2$  *непосредственно достижима* из разметки  $M_1$ , и записывают это как  $M_1[Y]M_2$ .

Конечная или бесконечная последовательность шагов, а также отношение достижимости определяются и обозначаются так же, как и для стандартных сетей Петри (см. выше).

Замечание по реализации. Надо признать, что поиск кандидатов связываний для срабатывания является в общем случае неразрешимой задачей: выражения могут содержать общерекурсивные функции, для которых требуется найти аргументы по заданным значениям. Но есть хорошо определенный класс выражений, для которых эта задача разрешима: это образцы. Переменная или константа это образец, образцом является структура из образцов, а также список образцов. Важно (это проверяется автоматически), чтобы на части входных дуг стояли образцы, и чтобы в них встречалась каждая переменная из  $Var(t)$ , не считая лишь те переменные, которые встречаются только на выходных дугах. Для последних при срабатывании порождается случайное значение из ее типа, число элементов которого должно быть невелико.

Второй способ – через сведение к стандартной сети Петри

Для заданной CPN определим  $SPN = (\acute{P}, \acute{T}, \acute{A}, \acute{N}, \acute{I})$  следующим образом.

Множество мест  $\acute{P} = \{(p, v) | p \in P, v \in C(p)\}$ . Иначе говоря, каждое место  $p$  превращается в семейство мест, индексированное типом  $C(p)$ .

Множество переходов  $\acute{T} = \{(t, b) | t \in T, b \in B(t)\}$ . Каждый переход  $t$  превращается в семейство переходов, индексированное всевозможными связываниями  $b \in B(t)$  (не забудем, что в понятие связывания уже включено условие  $G(t)\langle b \rangle = true$ ).

Пусть  $a$  – входная/выходная дуга,  $N(a) = (p, t)/(t, p)$ . Она превращается в семейство входных/выходных дуг, индексированных всевозможными связываниями  $b \in B(t)$ . При этом  $\acute{N}((a, b)) = (\acute{p}, \acute{t})/(\acute{t}, \acute{p})$ , где  $\acute{p} = (p, Expr(a)\langle b \rangle)$ ,  $\acute{t} = (t, b)$ .

Начальная разметка:  $\dot{I}(\dot{p}) = \dot{I}(p, v) = (I(p)(\langle \rangle))(v)$ , где  $v \in C(p)$ , а  $\langle \rangle$  – пустое связывание.

Для всякой разметки  $M(p)$  для CPN соответствующая ей разметка стандартной сети есть  $\dot{M}(\dot{p}) = \dot{M}(p, v) = M(p)(v)$  для всех  $p \in P$  и  $v \in C(p)$ . Легко видеть, что это изоморфизм разметок,  $\mu : M \leftrightarrow \dot{M}$  (который есть не что иное, как «закарривание» и «раскарривание» функций). Понятия шага для CPN и SPN попросту совпадают. Одинаковость преобразований разметок, производимых шагом, вытекает непосредственно из определений. Поэтому совпадают и отношения достижимости. Этим фактически доказана

**ТЕОРЕМА 1.** *Два указанных выше способа определения семантики цветных сетей Петри эквивалентны.*

### 1.3. Пример

Приведем простой пример, иллюстрирующий богатые возможности CPN как средства описания алгоритмов. На рисунке 1 показана CPN-сеть, реализующая своеобразный вариант параллельной сортировки.

```
colset NxR = product INT*REAL timed;
fun InpArr(k,n)=if n<k then []
  else 1' (k,InpElem(k,n))+InpArr(k+1,n);
fun InpElem(k,n)=uniform(1.0,10.0);
var a,b:REAL;
var i,j:INT;
```

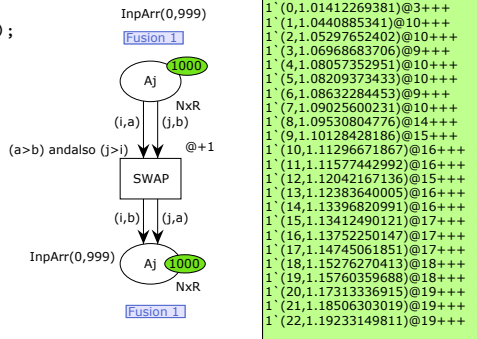


Рисунок 1. Пример: CPN-сеть, реализующая алгоритм сортировки

Это вполне работающий параллельный алгоритм, выполняемый за  $O(\log N)$  шагов<sup>2</sup>. Он состоит из одного места  $A_j$  и одного перехода SWAP. На рисунке мы видим два места, но это просто два изображения одного и того же места, о чем свидетельствует подпись Fusion 1 (для

<sup>2</sup>Это установлено эмпирически: с каждым удвоением размера массива число шагов увеличивается на 3-5 единиц (в прямой зависимости от начальной упорядоченности). В данном прогоне было выполнено 12420 переходов за 33 временных шага.



удобства). Слева вверху – сопутствующие определения на CPN ML. Место  $A_j$  имеет тип (colset)  $N \times R$ , являющийся произведением  $INT$  на  $REAL$ . Массив вещественных чисел представляется в виде мультимножества таких пар, у которых компонент  $INT$  кодирует номер (индекс) в массиве, а  $REAL$  – собственно значение. Две функции,  $InpArr$  и  $InpElem$ , служат для генерации случайного начального массива заданного размера. Соответствующий вызов  $InpArr(0,999)$ , порождающий 1000 элементов, приписан месту  $A_j$ . Сортировка заключается в подмене номеров в соответствии с порядком значений. На рисунке 1 справа показан начальный отрезок массива после завершения работы.

Переход  $SWAP$  соединен с местом  $A_j$  двумя входными и двумя выходными дугами. Согласно заданным на дугах выражениям переход срабатывает, выдергивая из места какие-либо два элемента, и тут же возвращает их, переставляя индексы. При переходе указана охрана:  $(a>b) \text{ and } also(j>i)$ , распознающая нарушение порядка. За один шаг могут одновременно сработать сразу несколько экземпляров перехода, при условии, что они не используют общих входных токенов. Поскольку каждый переход выполняется за время 1, на каждом такте выполняется шаг, состоящий из максимального множества непересекающихся готовых пар.

Все используемые переменные  $(a, b, i, j)$  и их типы заранее объявлены. Тип места задан как  $timed$ , чтобы моделировалось время работы. При этом переходу приписана задержка  $@+1$ , то есть 1 такт. С учетом всяких случайностей алгоритм сортирует 1000 элементов за 30–40 модельных тактов, выполняя в общей сложности около 13000 элементарных (попарных) обменов. Реальное же время работы интерпретатора – около 30 секунд, причем, чем ближе к концу, тем дольше идет подбор пар для очередного такта.

Существенный и почти неустранимый недостаток данного решения – отсутствие признака завершения. Второй дефект – оно нереализуемо «в железе». То есть непонятно, моделью какого реального процесса оно может быть. Здесь оно использовано просто для иллюстрации возможностей CPN. Как видим, они заметно шире, чем это нужно для описания (моделей) «реальных» систем.

## 2. Графический язык UPL

### 2.1. UPL как результат перевода из CPN

UPL – графический язык описания параллельных алгоритмов в парадигме потока данных (dataflow). Он имеет много общего с CPN. Вначале покажем, как UPL может быть получен в результате небольших трансформаций CPN при соблюдении некоторых ограничений возможностей CPN. Налагаемые ограничения будут помечаться буквами и **выделяться**. В подразделе 2.2 будет дано независимое определение графического языка UPL.

Первое и самое существенное ограничение – в исходной CPN-схеме **(А)** **из каждого места исходит ровно одна дуга.**

С точки зрения теории сетей Петри это очень сильное ограничение. Оно запрещает конфликты (между разными переходами), которые составляют основу сетей Петри. Правда, сохраняются конфликты между разными связываниями одного перехода (которые переходят в разные переходы стандартной сети Петри при описанном выше сведении). В дальнейшем это ограничение частично будет ослаблено через механизм ветвей языка UPL [11], что будет подробно рассматриваться в другой (будущей) статье.

Теперь мы можем избавиться от входных дуг на схеме, просто «приклеив» входные места к переходам, которые теперь будем называть *узлами*, а их входные места – *входами* узлов. Дуги из переходов в места остаются в UPL и ведут из узла на вход другого (или своего же) узла. Теперь будем называть их *стрелками*.

Входу в UPL приписывается локальное (для узла) имя и тип. Предполагается, что в исходной CPN-схеме **(В)** **выражение на входной дуге было просто переменной или структурой из переменных.** Будем называть такие выражения *форматными*. В частности, они должны быть одноэлементными (как мультимножества). Они являются образцами частного вида, которые не накладывают ограничения на тип, а только обеспечивают удобный доступ к его частям.

На выходных дугах выражения могут быть любыми, но они могут подвергаться преобразованиям. Условное выражение распадается на две стрелки, с соответствующими условиями на них. Пустая альтернатива ликвидируется. Выражения-мультимножества также распадаются на несколько стрелок, но одиночное кратное значение переходит в одну стрелку с кратным токеном. Новые переменные (не использованные на входной дуге) заменяются явным вызовом генератора случайных значений.

Следующее важное ограничение – **(С)** **допускаются только охраны типа равенства между входными переменными на разных дугах.** Некоторые входные переменные могут встречаться повторно на разных дугах, что равносильно использованию охранного равенства между переменными. Заметим, что пример на рисунке 1 не может быть переведен в UPL из-за несоблюдения ограничения (А) и (С).

Переменные, использованные в охранных равенствах (или повторно), в UPL составляют индекс и выносятся в отдельную структуру, заключаемую в угловые скобки (на графической схеме – в шестиугольники). Все остальные переменные считаются элементами данных и должны быть уникальными среди выражений на входных дугах данного перехода. Структура индекса  $\langle i_1, \dots, i_k \rangle$  формируется из повторных переменных выражений на входных дугах перехода  $N$  и является общей как для узла  $N$  в целом, так и

для всех его входов. Пусть входное место  $p$  перехода  $N$  в CPN стало входом  $p$  узла  $N$  в UPL. Тогда каждая дуга, идущая в место  $p$  в CPN, превращается в стрелку, идущую на вход  $p$  в UPL, а стоящее на ней выражение  $E$  распадается на выражение-значение  $e$  и выражения-индексы  $\langle e_1, \dots, e_k \rangle$ . Компоненты выделяются путем сопоставления выражения  $E$  с форматным выражением  $F$  на дуге из места  $p$  в переход  $N$ . Компоненты, сопоставившиеся индексным переменным  $i_j$ , переходят в поля индекса  $e_j$ , оставшиеся – в значение входа  $e$ . Но не все индексные переменные  $i_j$  могут содержаться в  $F$ . Тогда на позиции отсутствующих в  $F$  переменных индекса ставится знак «\*», который понимается как «любой».

Теперь вместо поиска связывания перехода  $N$ , которое делает переход в CPN активным, в UPL будем искать набор токенов, по одному на каждом входе, имеющим ровно один общий индекс. В результате срабатывания узла выбранные токены с его входов снимаются (токен, имеющий кратность, сохраняется с уменьшенной на 1 кратностью), а по всем его выходным стрелкам формируются и передаются токены на входы других узлов.

На рисунке 2а представлен условный фрагмент CPN-сети, содержащий допустимый переход  $N$  общего вида, и на рисунке 2б – его преобразование в язык UPL согласно описанным выше правилам.

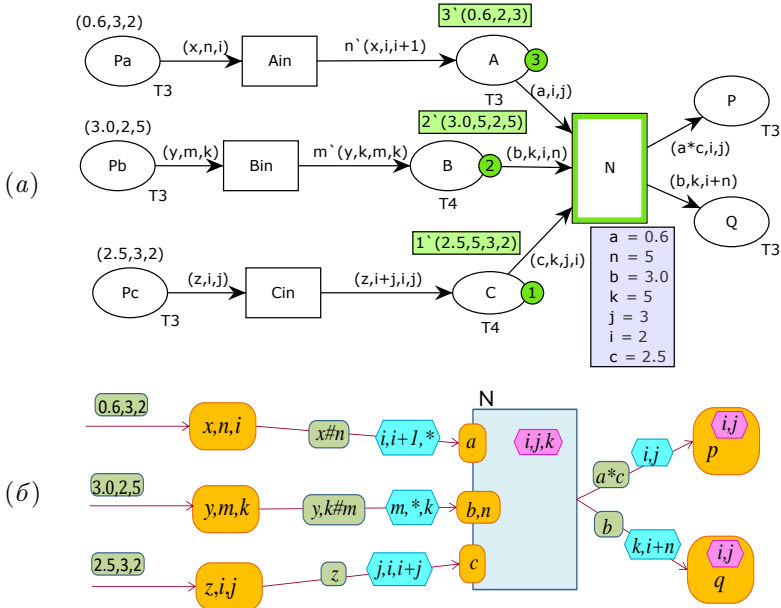


Рисунок 2. Преобразование типового перехода (N) сети CPN (a) в узел N схемы UPL (б)

Переход  $N$  имеет три входных места  $A, B, C$  типов  $T3, T4, T4$  (записей из 3 или 4 элементов, соответственно). Среди переменных перехода  $N$  три переменные:  $i, j, k$  – используются повторно, поэтому они станут индексом. Остальные переменные:  $a, b, n, c$  – в схеме на UPL будут входными данными трех входов. Подающие переходы  $Ain$  и  $Bin$  порождают кратные токены,  $Cin$  – одиночный. На схеме изображено состояние непосредственно перед срабатыванием перехода  $N$ . Под ним изображено связывание, с которым переход  $N$  готов к срабатыванию. После срабатывания место  $C$  станет пустым, а на местах  $A$  и  $B$  останутся токены с кратностями 2 и 1 соответственно.

Результирующий узел  $N$  изображен в виде прямоугольного блока. В нем имеется индекс  $\langle i, j, k \rangle$ , показанный в розовом шестиугольнике, и три входа:  $a, "b, n"$ , и  $c$ . Подающие переходы  $Ain, Bin, Cin$  превратились в три одноходовых узла без индексов, которые изображаются без прямоугольников. В качестве имен (входов) фигурируют списки входных переменных.

Первые два подающих узла выдают токены с кратностями, стоящими после знака  $\#$ . Входные данные на каждой дуге превратились в данные с индексами. Между индексом токена на стрелке и индексом узла имеется соответствие согласно позициям. На места недостающих во входном выражении переменных ставится знак «\*», который является «джокером». При сопоставлении он может принять любое значение. С учетом заданных значений будут поданы токены:

$$0.6\#3 \rightarrow N.a\langle 2, 3, * \rangle$$

$$3.0\#2 \rightarrow N.(b, n)\langle 2, *, 5 \rangle$$

$$2.5 \rightarrow N.a\langle 2, 3, 5 \rangle$$

Данная тройка образует активную комбинацию с единственным общим индексом  $\langle 2, 3, 5 \rangle$ . Джокер в первом токене примет значение 5, во втором – 3. В результате срабатывания узла  $N\langle 2, 3, 5 \rangle$  на двух первых входах останутся токены с кратностями 2 и 1, на третьем – ничего.

Примечание. Выходные места  $P$  и  $Q$  здесь превращены в одноходовые узлы с индексами, хотя вводить индексы здесь не было необходимости, поскольку это разные узлы, а не входы одного узла.

Использованные здесь обозначения и правила поясняются в следующем разделе. Забегая вперед, укажем на еще одно необходимое ограничение CPN, которое в данном примере соблюдено: в результате перевода в UPL **(D) хотя бы на одном входе должен быть токен с кратностью 1.**

В противном случае может произойти семантическая путаница (см. пояснение к примеру 2 ниже). Пока отметим только, что (в отличие от CPN) в UPL токен с кратностью, вообще говоря, *не* равносителен соответствующему количеству одинаковых экземпляров однократного токена.

Рассмотренные преобразования с сопутствующими им ограничениями собраны в таблице 1.

ТАБЛИЦА 1. Сводная таблица соответствия (преобразований) CPN→UPL с сопутствующими ограничениями

Преобразование CPN → UPL	Ограничения CPN	Ограничения UPL
Входные места «приклеиваются» к переходу в виде входного порта.	(A) Из каждого места исходит лишь одна дуга, (B) на ней выражение форматное.	
Условие в выражении на выходной дуге переходит в условие выдачи токена. Альтернативы и суммы распадаются на разные стрелки. Новые переменные на выходных дугах заменяются явным вызовом генератора случайных значений.	После предварительных преобразований выходные дуги перехода несут одноэлементные (как множества) выражения, возможно с кратностью.	
Отождествляемые (через охраны или повторные переменные) части цвета собираются в индекс.	(D) Охраны: только равенства переменных (включая повторные переменные) разных дуг.	
В выходном токене на место недостающей переменной индекса ставится «*».		
Кратность значения на выходной дуге переходит в кратность токена	(D) Хотя бы на одно входное место перехода должны приходиться только токены кратности 1.	Хотя бы на один вход узла приходят только токены кратности 1.

## 2.2. Независимое определение UPL

В этом подразделе дается независимое (от CPN) определение языка UPL и его семантики.

Программа на языке UPL раскладывается на отдельные узлы, которые передают друг другу сообщения – токены.

Узел имеет *имя*, один или несколько именованных *входов* и *индекс*, которым различаются экземпляры узлов. Индекс обычно имеет тип

структуры с целочисленными полями. Входные токены приходят на вход узла и могут на нем накапливаться. Узел в программе изображается блоком, входы – небольшими кружками на его стороне. Узел определяет класс *конкретных* узлов, характеризующихся конкретным значением индекса. Конкретный узел срабатывает, когда на каждом входе для него имеется токен. Этот набор токенов называется *активной комбинацией* (АК), а срабатывание – *активацией*.

При срабатывании участвующие в АК токены обычно одновременно снимаются с входов или, если у токена есть кратность, остаются на нем с уменьшенной на 1 кратностью. Оставшийся токен затем может участвовать в создании других АК с другими токенами. Но: никакая АК не может сработать повторно. Это принцип *однократной активации* (ПОА). Он имеет значение, когда все токены АК имели кратность. Это допустимо в UPL и полезно для организации взаимодействий по типу «каждый с каждым».

Обычно требуется, чтобы по приходу каждого токена на один из входов срабатывали (в некотором порядке) все новые АК. Мы называем это принципом *последовательной активации* (ППА). Но это не исключает параллелизм. Если две (или более) АК не пересекаются (нет общих токенов), они могут сработать одновременно. Это позволяет разносить изолированные подмножества токенов на разные процессоры (ядра), которые внутри работают последовательно, а между собой параллельно.

Узел-класс также имеет *программу*, которая выполняется (возможно, с некоторой задержкой) после срабатывания АК.

Программа узла пишется на некотором процедурном языке программирования (С, Паскаль и т.п.). В ней в качестве аргументов могут использоваться только входные данные (из участвующих в АК токенов) и поля индекса. Наряду с обычными операторами языка программа узла может содержать операторы отправки токена вида:

$$D[\#m] \rightarrow N.p\langle I \rangle,$$

где  $D$  – выражение, вырабатывающее значение, кратность  $m$  – целочисленное выражение или «#»,  $N$  – имя узла,  $p$  – имя входа, индекс  $I$  – список целочисленных выражений или символов «\*» (через запятую). Квадратные скобки здесь это метасимволы, указывающие на возможность отсутствия (когда кратность равна 1). Запись «##» понимается как бесконечная кратность. Оператор отправки токена читается: послать значение  $D$  в виде токена кратности  $m$  на вход  $p$  узла  $N$  с индексом  $I$ .

Эту же запись (с вычисленными выражениями) можно рассматривать как формат самого токена.

В графической форме программа записывается внутри блока (обычно это набор присваиваний локальным переменным, завершаемый выражением, вырабатывающим посылаемое значение) а каждый оператор посылки изображается стрелкой от границы блока к входу узла. Формат индекса задается в (розовом) шестиугольнике внутри блока. На стрелке записываются: у основания – условие посылки, в середине в округленном прямоугльнике – посылаемое значение как выражение, ближе к концу в шестиугольнике – целевой индекс (структура из выражений или «\*»). Все выражения обрабатываются в контексте узла-отправителя в терминах его имен входов, полей индекса и локальных переменных.

Результатом работы программы узла являются сформированные в ней токены. Весь внешний эффект программы узла состоит только в порождении некоторого числа новых токенов. По завершении своей программы конкретный узел прекращает свою активность, но может быть вновь активирован новым набором токенов.

В общем случае индекс в токене может содержать «\*» на месте некоторых полей, и тогда его адресатом считается не один конкретный узел, а сразу множество конкретных узлов, индекс которых в этой позиции имеет любое значение. Пусть узел  $N$  имеет  $s$  входов:  $p_1, \dots, p_s$ . Тогда набор из  $s$  токенов, направленных на все различные входы узла  $N$  будет активной комбинацией (АК), если их индексы  $I_1, \dots, I_s$  как множества имеют непустое, и притом одноэлементное пересечение  $I_0$ . Иначе говоря, когда имеется единственный конкретный узел  $N(I_0)$ , входящий в множества адресатов каждого токена из данной АК. (Ситуация, когда такой узел не единственный, считается ошибкой, которая вызывает аварийное прекращение работы.) Поэтому активироваться может только конкретный узел, а не их множество. Мы называем это принципом конкретности активации (ПКА).

Все выражения в программе узла могут использовать в качестве своих аргументов только локальные переменные, имена входов и имена полей индекса. Есть еще доступ к глобальным константам, которые принимают значение перед началом работы программы и в процессе ее не изменяются. Таким образом, множество и содержимое исходящих (из узла с конкретным индексом) токенов может зависеть только от информации, содержащейся во входящих (в данный конкретный узел) токенах.

### 3. Примеры простых фрагментов программ на UPL и CPN

На рисунке 3 приведены несколько примеров простых операций линейной алгебры над скалярами и векторами. Слева показана схема

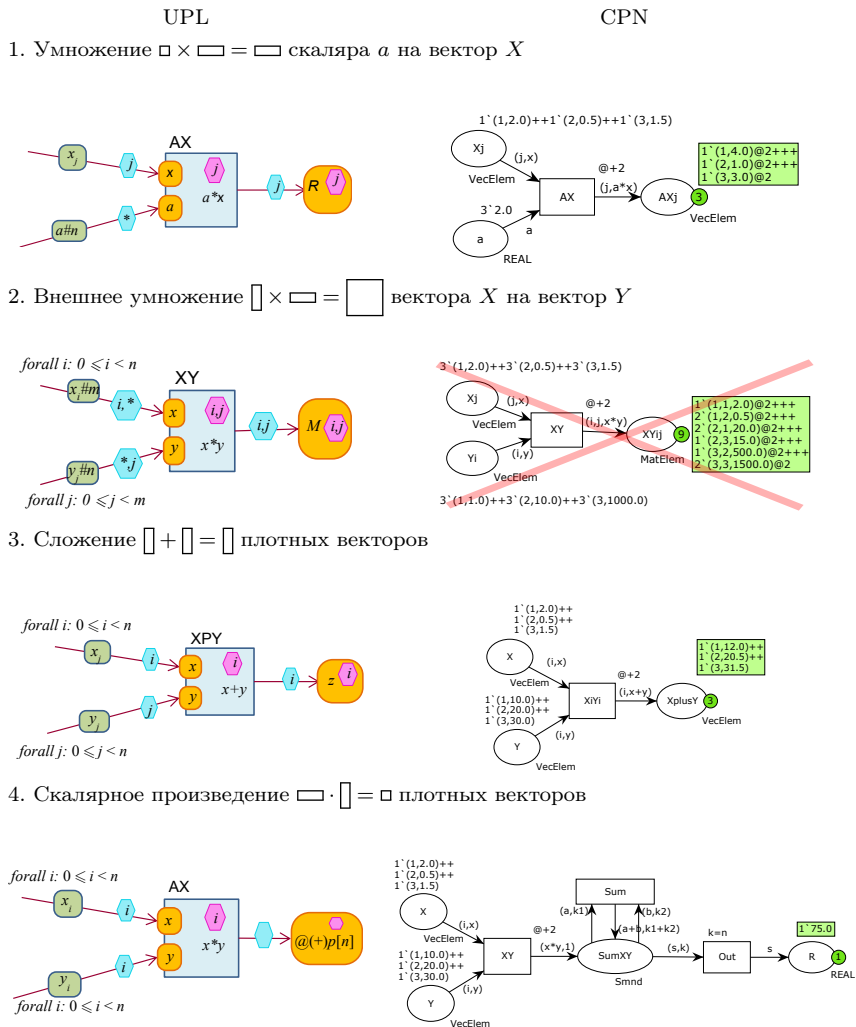


Рисунок 3. Примеры простых программ на UPL (слева) и их «перевод» в CPN (справа)



на UPL, справа – ее перевод на CPN. В обоих вариантах токены содержат только скалярные данные, поэтому векторы и матрицы представляются как множества токенов, содержащих значения и индексы.

Все схемы являются не законченными программами, а их фрагментами. Поэтому в схемах на UPL входные стрелки имеют свободное начало, а выражения на них условны. По некоторым из них придут множество токенов, что отражено в комментарии forall. (Вообще, все надписи вне объектов в UPL считаются комментариями в UPL, включая имена узлов). А выходные стрелки фрагментов упираются в условные одновходовые транзитные узлы, изображаемые как «вход без узла». В CPN этой «проблемы» нет, поскольку дуги всегда идут либо из места в переход, либо из перехода в место, и все необходимые интерфейсные места на схемах присутствуют.

Далее приводятся пояснения по каждому фрагменту.

#### (1) Умножение $n$ -вектора $X$ на число $a$

UPL. На вход  $x$  узла  $Ax$  приходят элементы вектора (в количестве  $n$  штук), а на вход  $a$  – один токен с общим множителем, индексом  $\langle * \rangle$  и кратностью  $n$ . (Здесь и далее  $n$  – константа, определенная глобально). В узле выполняется операция умножения множителя  $a$  на каждый отдельный элемент  $x$ . Произведение посылается как значение на выход (здесь оно подразумевается по умолчанию) с индексом  $j$ . (Здесь индекс на выходной стрелке тоже можно было бы не писать, поскольку он совпадает текстуально с индексом целевого узла  $R$ ). Это простой групповой узел.

CPN. Входные места  $X_j$  и  $a$  принимают, соответственно, набор пар  $(j, x)$  и множитель с кратностью, равной количеству этих пар. Каждая пара создает срабатывание, от которого на выход идет пара  $(j, a*x)$ .

#### (2) Умножение (внешнее) $n$ -вектора $X$ на $n$ -вектор $Y$

UPL. На оба входа поступают элементы векторов, причем кратность каждого элемента равна (или больше) числу элементов на другом входе. Согласно правилам происходит взаимодействие по типу «каждый с каждым». Обратите внимание на «\*» в индексах. В узле аргументы  $x$  и  $y$  перемножаются, и выдаются произведения с «приклеенными» двумя индексами. Это двойной групповой узел. Вектора могут быть разреженными, важно только, чтобы кратности соответствовали числу токенов на противоположном входе.

CPN. Если выполнить перевод по обычным правилам, ничего хорошего не получится. Поскольку на обоих входных местах присутствуют кратные элементы, то ничто не мешает срабатывание

с ними повторить несколько раз, в пределах меньшей из кратностей. В этом проявляется различие семантик кратности: в CPN она равносильна соответствующему количеству копий токена, а в UPL токен с кратностью это один токен, но с атрибутом кратности. Правила взаимодействия в UPL запрещают повторную активацию одной и той же комбинации токенов. В логике CPN такого нет и быть не может, поскольку кратность есть просто сокращенное обозначение для нескольких одинаковых токенов. Приходится признать, что в CPN отсутствует непосредственный аналог двойным групповым узлам, имеющим большое прикладное значение (умножением матриц, задача «N тел» и т.п.). Проблему их моделирования в CPN будем рассматривать в другой статье.

- (3) **Сложение плотных векторов UPL.** Здесь так же на оба входа приходят вектора, но, во-первых, их размеры и множества индексов должны совпадать, а, во-вторых, срабатывают только пары с равными индексами. И не важно, что на стрелке индекс токена  $u$  назван  $j$  (это просто выражение, вычисляемое в контексте узла-отправителя), важно, что он стоит в нужной позиции, которая в целевом узле имеет имя  $i$ . Важно также, что оба вектора плотные: если какой-то элемент в одном векторе будет отсутствовать, то соответствующий элемент «зависнет» на другом входе.

CPN. Перевод прямолинейный и адекватный. Обращаем внимание на одинаковые индексы  $i$  на обоих входных стрелках.

- (4) **Скалярное произведение плотных векторов**

UPL. Здесь также на входах два плотных вектора с одинаковым числом элементов, но их соответственные элементы теперь перемножаются и произведения передаются на суммирующий узел  $r$  без индекса. Префикс имени  $@(+)$  указывает на использование редукции сложением. Суффикс имени  $[n]$  говорит, что должно прийти ровно  $n$  слагаемых, после чего узел сработает. Здесь  $n$  — это глобальная константа (также может стоять число). Переменная в квадратных скобках также может быть именем другого входа — тогда число ожидаемых слагаемых определяется динамически. Ниже будет пример такого использования.

CPN. Здесь пришлось описать конкретный способ накопления суммы. Произведения (слагаемые)  $s$  посылаются на место  $\text{SumXY}$  в виде пар  $(s, k)$  с приклеенным счетчиком  $k = 1$ . В общем случае пара  $(s, k)$  означает, что  $s$  является суммой  $k$  слагаемых. Если в данном месте появляются два токена такого вида, то они покомпонентно складываются (переход  $\text{Sum}$ ) и сумма возвращается обратно на то же

место. Когда счетчик станет равен  $n$ , сработает переход `Out`, который положит итоговую сумму на место `R`.

Все рассмотренные примеры схем на языке CPN были проиграны в *CPN Tools*<sup>(URL)</sup>. Начальные данные можно видеть непосредственно над входными местами, а результаты – в зеленых прямоугольниках возле выходных мест. К сожалению, непосредственной реализации языка UPL в настоящее время нет. (Но есть эмулятор предшествующего языка DFL с ограниченными возможностями, реализованный на модели мультипроцессора ППВС «БУРАН» [11]).

Интересно выразить пример 4 на UPL без использования суммирующего входа. К сожалению, сделать это по аналогии с вариантом на CPN не получится, поскольку там одно место `SumXY` используется двумя переходами, а переходом `Sum` даже дважды. К тому же второй переход `Out` имеет охрану общего вида ( $k=n$ ), а это сравнение переменной и константы, а не между переменными. Самый простой вариант изображен на рисунке 4. Здесь пришлось дополнительно посылать начальный токен

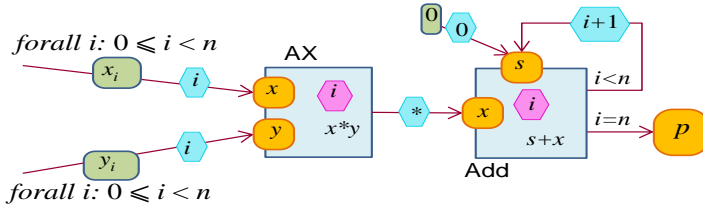


Рисунок 4. Вариант примера 4 без собирающих узлов

$0\langle 0 \rangle$  на вход `s`, куда также подается каждая следующая сумма  $(s+x)\langle i+1 \rangle$  при  $i < n$ . Та же сумма при  $i = n$  посылается на выходной порт `p` (без индекса). Заметим, что слагаемые подаются с индексом  $\langle * \rangle$ , что ведет к недетерминированному порядку суммирования.

#### 4. Пример: решение треугольной СЛАУ

Теперь рассмотрим более цельный пример алгоритма, а именно – решатель треугольных СЛАУ с разреженной матрицей. Он может применяться как ядро итерационного решателя СЛАУ общего вида методом Гаусса-Зейделя. Алгоритм ядра решателя [12] показан на рисунке 5.

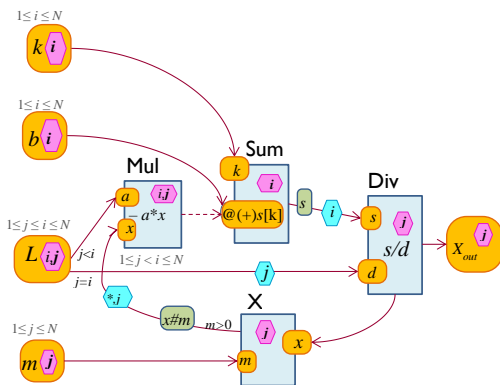


РИСУНОК 5. Алгоритм решения треугольной СЛАУ на UPL

Требуется вычислить вектор  $x$  из уравнения  $Lx = b$ , где матрица  $L$  – нижнетреугольная, то есть все ее элементы выше диагонали нулевые. При этом все диагональные элементы  $L_{ii}$  ненулевые. Решение вычисляется по формуле (1):

$$(1) \quad x_i = (b_i - \sum_{j=1}^{i-1} x_j L_{ij}) / L_{ii} \quad \text{для } i \text{ от } 1 \text{ до } N$$

Согласно букве формулы (1) вектор  $X$  вычисляется через себя. Однако, благодаря треугольности матрицы  $L$  зависимости между отдельными элементами ациклические. Матрица  $L$  может быть разреженной, и даже не быть треугольной, но сводимой к треугольной при некоторой синхронной перестановке строк и столбцов.

Алгоритм, изображенный на рисунке 5, реализует формулу (1) один к одному, не считая замены вычитания сложением. Каждая операция  $(+, *, /)$  становится самостоятельным узлом. Еще один узел (**X**) принимает вычисленное значение  $x_i$  и присоединяет к нему нужную кратность. Входные данные:

- $k_i$  – число ненулевых элементов в  $i$ -й строке;
- $b_i$  – элемент вектора правой части (плотного);
- $L_{ij}$  – ненулевой элемент матрицы;
- $m_j$  – число ненулевых элементов в  $j$ -м столбце без диагонального.

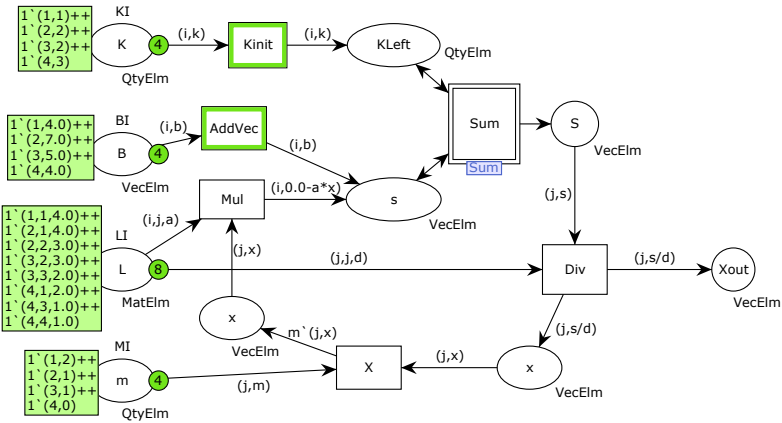
Суммирование производится собирающим входом  $@(+)\mathbf{s}[\mathbf{k}]$  узла **Sum**. Число слагаемых  $k$  приходит на другой вход этого же узла. Здесь оно всегда положительное, поскольку включает обязательный диагональный

элемент. (Если бы пришло нулевое  $k$ , то узел Sum сразу выдал бы нулевой результат, не дожидаясь и не используя слагаемых  $s$ ).

После деления суммы на диагональный элемент узлом Div частное  $x_i$  посылается на узел X. Тот приклеивает к нему кратность  $\#m_i$  и отправляет на умножитель Mul со звездочкой в позиции индекса  $i$ , то есть на весь столбец  $j$ . Если  $m = 0$ , токен не посылается.

Во всех узлах индексе  $i$  – номер строки – служит разделяющим контекстом, благодаря которому вычисления для разных строк происходят параллельно и не мешают друг другу.

Перевод в CPN, показанный на рисунке 6, имеет свои особенности.

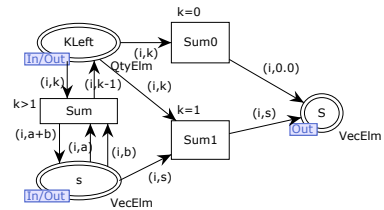


(a) верхний уровень

```

colset MatElm = product INT*INT*REAL;
colset QtyElm = product INT*INT;
colset VecElm = product INT*REAL;
var i,j,k,m:INT;
var a,b,d,x,s:REAL;
val BI=1'(1,4,0)++1'(2,7,0)++1'(3,5,0)++1'(4,4,0);
val KI=1'(1,1)++1'(2,2)++1'(3,2)++1'(4,3);
val LI=1'(1,1,4,0)++ 1'(2,1,4,0)++1'(2,2,3,0)++...
val MI=1'(1,2)++1'(2,1)++1'(3,1)++1'(4,0);
    
```

(б) декларации переменных, цветов и констант



(в) подсеть суммирования, задающая переход-подстановку Sum

Рисунок 6. Иерархическая CPN-сеть решения треугольной СЛАУ

Если в UPL выделение диагональных элементов реализуется условием ( $j=i$ ) у основания стрелки как частью программы узла, то в CPN оно задается образцом ( $j,j,d$ ) и реализуется как часть механизма

срабатывания перехода  $\text{Div}$ . На другой стрелке, исходящей из того же места  $L$ , образец  $(i, j, a)$  не накладывает условия  $i \neq j$ , но оно тут и не требуется, поскольку токен  $(j, x)$  в месте  $X$  (также входном для перехода  $\text{Mul}$ ) появится только после удаления диагонального элемента  $(j, j, d)$ .

Узел суммирования  $\text{Sum}$  переводится в CPN как переход-подстановка  $\text{Sum}$ . Он раскрывается статически в одноименную подсеть, описанную отдельно. Подсеть сообщается с основной сетью через свои интерфейсные места  $\text{Kleft}$ ,  $s$  и  $S$ , из которых первые два входно-выходные, последнее — чисто выходное. Соответственно, в подсети имеются дуги, ведущие из первых двух мест, и дуги, ведущие во все три места. Подсеть реализует конкретный способ суммирования, примерно как в примере 4 выше. Отличие в том, что число слагаемых поступает динамически. Оно приходит на место  $\text{Kleft}$ , в котором всегда будет находиться число оставшихся слагаемых. Слагаемые приходят в место  $s$ . Всякий раз, когда в нем обнаруживаются два слагаемых (с общим  $i$ ), они складываются переходом  $\text{Sum}$  и сумма возвращается туда же. Одновременно из  $\text{Kleft}$  вычитается 1. Когда  $\text{Kleft}$  достигнет 1, сработает переход  $\text{Sum1}$ , передающий результат из  $s$  на  $S$ . Также предусмотрено, что если придет  $\text{Kleft} = 0$ , то переход  $\text{Sum0}$  отреагирует выдачей суммы 0.0. Строго говоря, чтобы передать точную семантику суммирующего узла UPL, переходы  $\text{Sum1}$  и  $\text{Sum0}$  должны иметь более высокий (по сравнению с  $\text{Sum}$ ) приоритет (а такая возможность в CPN есть), но здесь, если все данные корректны, в этом нет необходимости. Перевод узлов  $\text{Div}$  и  $X$  прямолинеен:  $m$  также становится кратностью значения  $(j, x)$ . И хорошо, что на другом входе узла  $\text{Mul}$  в UPL находятся токены (элементы матрицы  $L_{ij}$ ) с полным индексом, и не возникает ситуации двойного группового узла.

## 5. Сходства и различия UPL и CPN

Узлы UPL соответствуют переходам в CPN, входы узлов — местам, токены (стрелки) — выходным дугам переходов. Как и в CPN, на входах токены могут накапливаться в виде мультимножеств. Конкретные узлы и активные комбинации соответствуют связываниям в CPN.

Есть сходство и в способах нахождения активных наборов. В обоих случаях имеется некоторое множество векторов вида  $I^m = \langle i_1, \dots, i_m \rangle$ , из которых нужно отобрать набор из  $k$  векторов, причем условием для образования набора является равенство некоторых компонентов между собой. Тем самым для поиска наборов требуется сопоставление и сравнение по содержимому. Правда, есть и отличие. В CPN могут использоваться вектора разного типа, размера (в разных местах) и сравниваться могут компоненты разных позиций (как одного вектора, так и разных). Это задается выражениями-образцами, а равенство компонент определяют

одинаковые имена. А в UPL вектора (индексы) в одном наборе должны быть одного размера, и на равенство проверяются компоненты разных векторов, находящиеся в одной позиции. При этом в самих векторах указывается для каждой позиции, используется ли данный компонент в сравнении.

Есть и другие отличия. В CPN при переходах может быть охрана – условие произвольного вида, проверяемое в рамках операции срабатывания. В UPL такие проверки могут производиться только в рамках программы узла, выполняемой после срабатывания АК, но не в рамках самого срабатывания.

Есть довольно тонкое семантическое различие в подборе и срабатывании группы АК в UPL и шага в CPN. В CPN токены, образованные на предыдущем срабатывании считаются сразу записанными в целевые места. При наличии тайминга (задержек на дугах или на переходах) это откладывается на некоторое время. Вообще, наличие времени говорит о назначении CPN как средства моделирования поведения. В UPL времени нет, поскольку это средство записи вычислительных алгоритмов. О времени можно говорить только в рамках моделирования работы «реальной» UPL-машины. Семантика самого UPL утверждает, что никакой токен не может быть доставлен мгновенно, какое-то положительное время это обязательно займет. Поэтому семантика UPL рассматривает состояние не как одно, а как два (мульти) множества токенов: входной буфер (ВБ) и рабочее пространство (РП). ВБ есть абстракция токенов, находящихся в движении к своей цели. РП – это токены, достигшие своих мест назначения на входах узлов. Все токены АК, готовой к срабатыванию, должны быть в РП. В теории из ВБ токены по-одному передаются в РП. И после каждой такой передачи ищутся все возникшие АК и срабатывают. Перед приемом нового токена из ВБ в РП (по крайней мере в теории) не должно оставаться АК.

Выше (раздел 2.2) мы назвали это принципом последовательной активации (ППА). Именно он позволяет без проблем обеспечивать соблюдение принципа однократной активации (ПОА) в ситуации, когда все токены АК имеют кратности больше 1. Действительно, по приходу одного из таких токенов, машина проводит поиск АК только с участием нового токена, и в рамках этого поиска каждая найденная АК срабатывает только один раз, каждый раз уменьшая кратности на 1. Работа с новым токеном прекращается, если его кратность упадет до 0. И только после срабатывания всех найденных АК новый токен заносится в РП с кратностью, уменьшенной на число срабатываний. Понятно, что при таком механизме, АК из тех же токенов уже не может сработать вновь, что и обеспечивает принцип однократности почти «бесплатно». При этом

важно, что токен с уменьшенной кратностью считается «тем же самым» токеном. И это радикально отличает ситуацию токена с кратностью от ситуации с заданным числом копий однократного токена. В CPN этого семантического различия нет, и потому нам приходится накладывать ограничение, чтобы хотя бы на один из входов токены всегда приходили с кратностью 1. А это не позволяет проводить взаимодействия по принципу «каждый с каждым».

Принцип последовательной активации кажется противоречащим параллелизму и распределенности. Однако, это не так. ППА соблюдается как принцип теории. На практике же виртуальное адресное пространство (ВАП) конкретных узлов может быть разбито на независимые части (с некоторыми оговорками) так, что каждый токен направляется в свою часть РП, локализованную в своем физическом процессоре (ядре). И хотя внутри каждого ядра поступающие токены обрабатываются строго последовательно по одному, разные ядра работают независимо и параллельно. В следующем разделе подробно описывается, как это достигается.

## 6. Функция распределения

В UPL глобальное адресное пространство определяется программой. В нем виртуальным адресом является пара (имя узла, конкретный индекс). Для распределения вычислений по вычислительным ядрам пользователь задает функцию распределения (ФР), которая отображает виртуальные адреса в номера ядер. ФР однозначно указывает для каждого конкретного узла номер ядра, в котором должно произойти его срабатывание.

Распределение конкретных узлов индуцирует соответствующее распределение токенов. Поэтому каждый токен, будучи создан в каком-либо ядре, направляется через коммуникационную сеть непосредственно в то ядро, где будут производиться срабатывания с его участием. Номер ядра определяется функцией распределения, которая вычисляется непосредственно после создания токена перед выдачей его в сеть. Будем называть его целевым адресом (токена) и обозначать как  $F(N, I)$  или  $F_N(I)$ , где  $N$  – имя узла, а  $I$  – индекс.

Чтобы целевой адрес токена  $F_N(I)$  был всегда однозначным, следует избегать использования в качестве аргумента функции  $F$  тех компонент индекса, которые в некоторых токенах, направляемых на узел  $N$  заданы как «\*». (В противном случае значением функции  $F$  для такого токена будет множество номеров ядер, представленное битовой строкой в алфавите  $\{0, 1, *\}$  [13]. Так, для решателя треугольной СЛАУ на рисунке 5 будет разумным использовать для всех узлов индекс  $j$  (а если его нет, как



у узла `Sum`, то `i`): на схеме он всюду выделен жирным шрифтом. Это будет разбиение «по столбцам».

В работе [12] эта задача рассматривается в специальной постановке, когда как номер уравнения  $i$ , так и номер переменной  $j$  являются трехмерными координатами элементов сетки  $(x, y, z)$ . При этом коэффициенты матрицы  $L$  ненулевые только для тех пар  $(i, j)$ , которые соответствуют соседним ячейкам сетки (отличающимся не более чем на 1 по каждой координате). Там была предложена следующая функция распределения  $F : (x, y, z) \rightarrow (p_1, p_2)$ :

```
F(x,y,z) =
  let
    val X =(x+y+2*z)/d
    val Y = (y+z)/d
    val Z = z/d
  in
    ( (K1*d*(X-Y-Z)/nx) mod K1,
      (K2*d*(Y-Z)/ny) mod K2 )
```

где  $K1, K2, d, nx, ny$  – константы, а  $X, Y, Z$  – блочные координаты. Она порождает нетривиальное разбиение адресного пространства на блоки, являющиеся косыми параллелепипедами, и последующее распределение блоков между ядрами. Оно оказалось выигрышным с точки зрения длины критического пути (цепочки зависимых передач между ядрами) и объема коммуникаций по сравнению с более простым распределением на прямоугольные блоки.

Здесь мы привели этот пример лишь как иллюстрацию того, что распределение вычислений между ядрами можно изменять в широких пределах, просто варьируя формулы функций распределения. Важно, что ФР задается отдельно от основного тела программы на UPL и не может «испортить» ее результат (если, конечно, программа правильная, и ФР корректна [13]), а может повлиять только на время ее работы. Выбор ФР, как правило, имеет целью улучшить следующие показатели работы программы:

- (1) Равномерность вычислительной нагрузки на ядра.
- (2) Объем коммуникационной нагрузки на сеть (на разных уровнях ее иерархии) и ее равномерность во времени.
- (3) Длина критического пути (задержка на цепочках передач между ядрами, где каждая следующая передача зависит от предыдущей).

Улучшение этих показателей может способствовать существенному повышению эффективности работы программы на распределенной системе. Что касается возможности распределенного выполнения CPN, пока эта проблематика не исследована. Есть много работ по проблеме

распределенной реализации обычных (не цветных) сетей Петри, причем почти всегда они либо очень сложны, либо не полны (не для любых сетей). Есть много статей о применении CPN для моделирования распределенных систем (фактически таковы почти все применения), но о распределенной реализации самого CPN речи нигде нет. Это вообще вряд ли возможно, если не накладывать какие-либо ограничения. Фактически в данной статье мы и накладываем ограничения (A)–(D), говоря о преобразовании из CPN в UPL. И, как показывает наш опыт использования UPL, они не создают проблем (кроме D) при описании вычислительных алгоритмов. Возможно, некоторые можно и не накладывать. Можно надеяться, что техника распределения UPL посредством функций распределения может быть адаптирована и к CPN при каких-то, возможно более слабых, ограничениях. Это требует дальнейшего изучения.

## 7. Близкие работы

Есть очень много работ по применению CPN для моделирования и анализа различных распределенных конкурентных (concurrent) систем. Однако, среди них почти не видно работ по применению CPN к анализу алгоритмов, тем более вычислительного типа. Редкое исключение – работа [14]. В ней CPN используется как инструмент моделирования параллельных алгоритмов, написанных в парадигме нескольких последовательных процессов, выполняемых на общей памяти с использованием средств обеспечения взаимного исключения. Из алгоритма извлекается CPN-сеть, моделирующая как последовательное выполнение каждого процесса, так и разделяемый доступ к общим ресурсам. Возможны разные степени абстракции, вплоть до полной симуляции вычислений. Исследуется пространство состояний моделей для обнаружения нежелательных состояний.

Наша работа отличается использованием языка описания алгоритмов, который по духу близок к CPN, и поэтому мы используем CPN не столько как средство анализа с целью верификации на разных уровнях абстракции (хотя это тоже возможно и полезно), сколько как средство симуляции и имплементации самого UPL. Также нам интересно сопоставление выразительных возможностей этих двух языков как средств описания вычислительных алгоритмов.

## Заключение

Были даны определения двух систем, CPN и UPL, достаточно точные для их сопоставления и сравнения как средств описания параллельных вычислительных алгоритмов. По-видимому, эта область применения,

в отличие от моделирования распределенных систем, налагает свои особые требования. В частности, полезна особая семантика кратности, допускающая взаимодействие множеств токенов по типу «каждый с каждым», а также специальные собирающие (в частности, суммирующие) места или входы. С другой стороны, некоторые черты CPN востребованы слабо: охраны общего вида, возможность иметь несколько конфликтующих стрелок из одного места.

Для обоих языков является удобным и наглядным графическое представление в виде графа, в котором блоки представляют действия (в CPN переходы, в UPL узлы), овалы – данные, а стрелки передачу данных. При этом в UPL входные места (порты) «приклеены» к узлам-действиям, что, на наш взгляд, отражает нужды языка вычислений, улучшая наглядность и снижая запутанность. Это также согласуется с тем, что, в отличие от CPN, в UPL каждое входное место принадлежит своему узлу-действию.

Семантика обоих языков основана на актах срабатывания действий, обусловленных готовностью для них входных данных и ничем иным. Одни действия создают данные, потребляемые другими действиями. Так возникает информационный граф между действиями. В литературе по CPN его называют *occurrence graph*. Ему близко понятие информационного графа алгоритма, введенного в [15]. В UPL это трасса вычислительного процесса, которая может быть визуализирована в виде графа (для отладки и анализа).

Особенностью UPL является разбиение входных данных узла на собственно данные и индекс, который присутствует, целиком или частично, в каждом входном токене. Только индекс используется при сопоставлении токенов с разных входов, и именно по индексу осуществляется распределение вычислений по «пространству». Для этой цели в распределенную (многопроцессорную) вычислительную систему (PBC), исполняющую UPL-программу, вводится возможность задавать, отдельно от программы, функцию распределения, зависящую от пары (имя узла, индекс), и вырабатывающую номер процессорного ядра PBC. Можно так же организовать и распределение по времени, способствующее уменьшению заполнения сопоставляющей памяти токенов. Вряд ли подобное можно ввести в CPN без наложения ограничений типа тех, что отличают UPL от CPN.









К сожалению, язык UPL пока не реализован. Имеются только реализации первоначального языка DFL, из которого UPL вырос как его обобщение. Все примеры пока просто набирались в PowerPoint, а для выполнения переводились вручную в DFL. Хотелось бы иметь аналог CPN Tools с графической формой входного языка для разработки и выполнения алгоритмов на UPL или близком к нему языке. Возможно








появление обобщенной рамочной системы графического программирования, частными случаями которой были бы как CPN, так и UPL.

В данной статье были рассмотрены только базовые средства языка UPL. Следующие элементы были только упомянуты и будут подробнее освещены в будущей статье (статьях):

- (1) Узлы с ветвями, реализующие конфликтные входы в рамках одного узла [7, 10].
- (2) Распределенные групповые узлы [14] в UPL.
- (3) Моделирование двойных групповых узлов в CPN.
- (4) Рекурсия и мемоизация.
- (5) Распознавание «типины».

### Список литературы

- [1] Petri C.A. *Kommunikation mit Automaten*, PhD thesis.– University of Bonn.– 1962.– 128 pp.  [↑](#)<sub>92</sub>
- [2] Котов В. Е. *Сети Петри*.– М.: Наука.– 1984.– 160 с. [↑](#)<sub>92</sub>
- [3] Orlov S. P., Susarev S. V., Uchaikin R. A. *Application of hierarchical colored Petri nets for technological facilities' maintenance process evaluation* // Applied Sciences.– 2021.– Vol. 11.– No. 11.– id. 5100.– 26 pp.  [↑](#)<sub>92</sub>
- [4] Shapiro R. M. *Validation of a VLSI chip using hierarchical coloured Petri nets* // Microelectronics Reliability.– 1991.– Vol. 31.– No. 4.– Pp. 607–625.  [↑](#)<sub>92</sub>
- [5] Jitmit C., Vatanawood W. *Simulating artificial neural network using hierarchical coloured Petri nets* // *Proceedings of the 2021 6th International Conference on Machine Learning Technologies, ICMLT 2021* (Jeju Island Republic of Korea, April 23–25, 2021), New York: ACM.– 2021.– ISBN 978-1-4503-8940-2.– Pp. 127–131.  [↑](#)<sub>92</sub>
- [6] K. Jensen *Coloured Petri nets: A high level language for system design and analysis* // *Advances in Petri Nets 1990, ICATPN 1989, Lecture Notes in Computer Science*.– vol. 483, Berlin–Heidelberg: Springer.– 1991.– ISBN 978-3-540-53863-9.– Pp. 342–416.  [↑](#)<sub>92</sub>, 95, 96
- [7] Климов А. В., Окунев А. С. *Графический потоковый метаязык для асинхронного распределенного программирования*, МЭС-2016 (Россия, Москва, октябрь 2016), Проблемы разработки перспективных микро- и наноэлектронных систем.– №2, М.: ИППМ РАН.– 2016.– С. 151–158.  [↑](#)<sub>92</sub>, 118
- [8] Климов А. В. *О парадигме универсального языка параллельного программирования* // *Языки программирования и компиляторы-2017, PLC-2017* (Южный федеральный университет, Институт математики, механики и компьютерных наук им. И. И. Воровича, 3–5 апреля 2017), Ростов-на-Дону: ЮФУ.– 2017.– ISBN 978-5-9275-2349-8.– С. 141–146.  [↑](#)<sub>92</sub>
- [9] Harper R. *Programming in Standard ML*.– Carnegie Mellon University.– 2011.– 297 pp.  [↑](#)<sub>95</sub>

- [10] Климов А. В., Левченко Н. Н. *Механизм ветвей в потоковом метаязыке UPL (METAL) и методы его реализации в ППВС «БУРАН»*, МЭС-2018 (Россия, Москва, октябрь 2018), Проблемы разработки перспективных микро- и нанoeлектронных систем.– №3, М.: ИППМ РАН.– 2018.– С. 31–37.  ↑118
- [11] Климов А. В., Левченко Н. Н., Окунев А. С., Стемпковский А. Л. *Вопросы применения и реализации потоковой модели вычислений*, МЭС-2016 (Россия, Москва, октябрь 2016), Проблемы разработки перспективных микро- и нанoeлектронных систем.– №2, М.: ИППМ РАН.– 2016.– С. 100–106.  ↑100, 109
- [12] Змеев Д. Н., Климов А. В., Окунев А. С., Левченко Н. Н. *Особенности реализации теста НРСГ для ППВС «БУРАН» // XXII Харитоновские тематические научные чтения* (онлайн, 24-27 мая 2021), Саров: РФЯЦ–ВНИИЭФ.– 2022.– ISBN 978-5-9515-0507-1.– С. 193–205.   ↑109, 115
- [13] Климов А. В. *Средства верификации распределения вычислений в потоковой архитектуре ППВС «Буран»*, МЭС-2020 (Россия, Москва, октябрь 2020), Проблемы разработки перспективных микро- и нанoeлектронных систем.– №4, М.: ИППМ РАН.– 2020.– С. 236–243.   ↑114, 115
- [14] Westergaard M. *Towards verifying parallel algorithms and programs using coloured Petri nets*, PNSE'2011 (Newcastle upon Tyne, UK, June 20-21, 2011), CEUR Workshop Proceedings.– vol. **723**.– 2011.– Pp. 57–71.  ↑116, 118
- [15] Воеводин В. В., Воеводин Вл. В. *Параллельные вычисления*.– СПб: БХВ-Петербург.– 2004.– ISBN 5-94157-160-7.– 599 с. ↑117

Поступила в редакцию	24.10.2023;
одобрена после рецензирования	26.11.2023;
принята к публикации	26.11.2023;
опубликована онлайн	14.12.2023.

Рекомендовал к публикации

*д.ф.-м.н. Н. Н. Непейвода*


### Информация об авторе:



Foto by A. Yu. Fomenko, CC-BY-SA

**Аркадий Валентинович Климов**

ст. научн. сотр. Института проблем проектирования в микроэлектронике РАН. Научные интересы: нетрадиционные модели параллельных вычислений

 0000-0002-7030-1517

**e-mail:** [arkady.klimov@gmail.com](mailto:arkady.klimov@gmail.com)

*Автор заявляет об отсутствии конфликта интересов.*



# Coloured petri nets and the language for distributed programming UPL: their comparison and translation

Arkady Valentinovich **Klimov**

Institute for Design Problems in Microelectronics

 [arkady.klimov@gmail.com](mailto:arkady.klimov@gmail.com)

**Abstract.** Petri nets are widely used as a means of modeling distributed multi-agent systems. There are tools for working with extended Petri nets, in which tokens are loaded with arbitrary data. For example, CPN Tools allows you to describe, play and explore Colored Petri Nets (CPN). The question is raised about the possibility of using this tool for the development, prototyping and research of parallel distributed computing algorithms, ideally turning them into working efficient parallel programs. We have experience in experimental programming of various algorithms in the graphical data flow language UPL, which currently exists “on paper”. Its comparison with CPN shows that their semantics have a lot in common. In the article, both languages are defined, compared with examples and through the rules of translation from one to another. The means for defining distribution of computations (distribution functions) are also described. An interesting question is about their transfer to CPN, where they have no analogues yet. (*In Russian*).

**Key words and phrases:** Petri nets, Coloured Petri Nets, parallel programming, dataflow computation model, algorithm graph, visual programming, UPL language, distribution function




2020 *Mathematics Subject Classification:* 68N15, 68N19; 97P40

**Acknowledgments:** The work is supported by Institute for Design Problems in Microelectronics

**For citation:** Arkady V. Klimov. *Coloured petri nets and the language for distributed programming UPL: their comparison and translation*. Program Systems: Theory and Applications, 2023, **14**:4(59), pp. 91–122. (*In Russ.*). [https://psta.psiras.ru/read/psta2023\\_4\\_91-122.pdf](https://psta.psiras.ru/read/psta2023_4_91-122.pdf)

## References

- [1] C.A. Petri. *Kommunikation mit Automaten*, PhD thesis, University of Bonn, 1962, 128 pp. [URL](#)
- [2] V. E. Kotov. *Petri Nets*, Nauka, M., 1984 (in Russian), 160 pp.
- [3] S. P. Orlov, S. V. Susarev, R. A. Uchaikin. “Application of hierarchical colored Petri nets for technological facilities’ maintenance process evaluation”, *Applied Sciences*, **11**:11 (2021), id. 5100, 26 pp. [doi](#)
- [4] R. M. Shapiro. “Validation of a VLSI chip using hierarchical coloured Petri nets”, *Microelectronics Reliability*, **31**:4 (1991), pp. 607–625. [doi](#)
- [5] C. Jitmit, W. Vatanawood. “Simulating artificial neural network using hierarchical coloured Petri nets”, *Proceedings of the 2021 6th International Conference on Machine Learning Technologies, ICMLT 2021* (Jeju Island Republic of Korea, April 23–25, 2021), ACM, New York, 2021, ISBN 978-1-4503-8940-2, pp. 127–131. [doi](#)
- [6] Jensen K. “Coloured Petri nets: A high level language for system design and analysis”, *Advances in Petri Nets 1990, ICATPN 1989, Lecture Notes in Computer Science*, vol. **483**, Springer, Berlin–Heidelberg, 1991, ISBN 978-3-540-53863-9, pp. 342–416. [doi](#)
- [7] A. V. Klimov, A. S. Okunev. “A graphical dataflow meta-language for asynchronous distributed programming”, MES-2016 (Rossiya, Moskva, oktyabr’ 2016), Problemy razrabotki perspektivnyx mikro- i nanoelektronnyx sistem, 2, IPPM RAN, M., 2016, pp. 151–158 (in Russian). [URL](#)
- [8] A. V. Klimov. “On the paradigm of a universal parallel programming language”, *Yazyki programmirovaniya i kompilyatory-2017, PLC-2017* (Yuzhnyj federal’nyj universitet, Institut matematiki, mexaniki i komp’yuternyx nauk im. I. I. Vorovicha, 3–5 aprelya 2017), YuFU, Rostov-na-Donu, 2017, ISBN 978-5-9275-2349-8, pp. 141–146 (in Russian). [URL](#)
- [9] R. Harper. *Programming in Standard ML*, Carnegie Mellon University, 2011, 297 pp. [URL](#)
- [10] A. V. Klimov, N. N. Levchenko. “Branches in the Dataflow Metalanguage UPL (METAL) and Methods of their Implementation in the PDCS ‘Buran’”, MES-2018 (Rossiya, Moskva, oktyabr’ 2018), Problemy razrabotki perspektivnyx mikro- i nanoelektronnyx sistem, 3, IPPM RAN, M., 2018, pp. 31–37 (in Russian). [doi](#)
- [11] A. V. Klimov, N. N. Levchenko, A. S. Okunev, A. L. Stempkovskij. “The application and implementation issues of dataflow computing system”, MES-2016 (Rossiya, Moskva, oktyabr’ 2016), Problemy razrabotki perspektivnyx mikro- i nanoelektronnyx sistem, 2, IPPM RAN, M., 2016, pp. 100–106 (in Russian). [URL](#)
- [12] D. N. Zmeev, A. V. Klimov, A. S. Okunev, N. N. Levchenko. “Features of HPCG benchmark implementation for the “BURAN” PDCS”, *XXII Xaritonovskie tematische skie nauchnye chteniya* (onlajn, 24–27 maya 2021), RFYaCz–VNIIEF, Sarov, 2022, ISBN 978-5-9515-0507-1, pp. 193–205 (in Russian). [doi](#) [URL](#)

- [13] A. V. Klimov. “Tools for Verification of Computation Distribution in the Parallel Dataflow Computing System (PDCS) ‘Buran’”, MES-2020 (Rossiya, Moskva, oktyabr’ 2020), Problemy razrabotki perspektivnyx mikro- i nanoelektronnyx sistem, 4, IPPM RAN, M., 2020, pp. 236–243 (in Russian).  
- [14] M. Westergaard. “Towards verifying parallel algorithms and programs using coloured Petri nets”, PNSE’2011 (Newcastle upon Tyne, UK, June 20-21, 2011), CEUR Workshop Proceedings, vol. **723**, 2011, pp. 57–71. 
- [15] V. V. Voevodin, Vl. V. Voevodin. *Parallel Computations*, BXV-Peterburg, SPb, 2004, ISBN 5-94157-160-7 (in Russian), 599 pp.