УДК 519.85(075.8)

ПОЛИМОРФНЫЕ СЕТЕВЫЕ МОДЕЛИ ДИСКРЕТНЫХ ПРОЦЕССОВ

М.П. Ревотюк
Белорусский государственный университет информатики и радиоэлектроники
Республика Беларусь, 220013, Минск, ул. П.Бровки, 6
E-mail: rmp@bsuir.unibel.by

Н.В. Хаджинова
Белорусский государственный университет информатики и радиоэлектроники
Республика Беларусь, 220013, Минск, ул. П.Бровки, 6
E-mail: kafitas@bsuir.unibel.by

Ключевые слова: дискретные процессы, сетевые модели, полиморфные классы, расширенные сети Петри, временные сети Петри
Key words: discrete processes, network models, polymorphic classes, extended Petri nets, timed Petri nets

Рассматривается представление сетевых моделей дискретных процессов шаблонами классов в терминах технологии объектно-ориентированного программирования. Иерархия полиморфных классов использована для определения сетей переходов. Рекуррентные алгоритмы интерпретации процессов построены на основе обобщения понятия временных сетей Петри.

POLYMORPHIC NETWORK MODELS OF DISCRETE PROCESSES / M.P. Revotyuk (Belarusian State University of Informatiks and Radioelectronics, 6 Brovki, Minsk 220013, Republic of Belarus, E-mail: rmp@bsuir.unibel.by), N.V. Khajunova (Belarusian State University of Informatics and Radioelectronics, 6 Brovki, Minsk 220013, Republic of Belarus, E-mail: kafitas@bsuir.unibel.by). Object of consideration - representation of network models of discrete processes by patterns of classes in terms of technology object-oriented programming. The hierarchy of polymorphic classes for definition of transition nets is used. Interpretation of processes recurrent algorithm constructed on idea Timed Petri Nets generalization.

1. Введение

В настоящее время высокий уровень спецификации систем управления дискетными процессами рассматривается чаще всего в терминах конечных автоматов или сетей Петри[1,2]. Конечные автоматы широко используются для построения систем логического управления как аппаратными, так и программными средствами [3]. Близость к булевой алгебре и теории графов, наглядность графического представления и детерминированность поведения являются заметными достоинствами этой абстракции. Недостатком конечных автоматов является их применимость для сугубо последовательных систем.

Сети Петри и их расширения занимают промежуточное положение между машинами Тьюринга и конечными автоматами. Привлекательность и эффективность использования сетей Петри объясняется тем, что сеть Петри – интеграция графа и дискретной динамической системы, может служить статической и динамической моделью представляемого объекта. Особенную роль сети Петри играют при моделировании асинхронных параллельных процессов. Наглядность динамики и композиционные возможности сетей Петри выше, чем у конечных автоматов, а компактность представления превосходит машину Тьюринга. Од-

нако при моделировании процесса управления сложными иерархическими системами, требующего проведения детализации отдельных фаз с целью поиска оптимальных решений возникает необходимость расширения понятия классических сетей Петри.

Расширения сетей Петри формально можно рассматривать в терминах определений специфических вариантов систем вида “условие-действие”(СУД). Аналитические возможности расширенных сетей Петри по мере усложнения расширений снижаются [1]. Однако такие модели оказываются удобными для спецификации системы для ее аппаратной или программной реализации [3], а также анализа систем методами имитационного моделирования [4,7]. При этом вне рамок модели остается процесс поиска закона управления.

При разработке систем управления в общем случае необходимо наличие моделей для решения задач анализа, синтеза и реализации управляющей части. Методы решения таких задач опираются на модели разной степени агрегирования состояний и часто остаются несовместимыми. Вопрос выбора единой формальной абстракции представления, способной отразить логическую схему системы на организационный и организационно-технологический уровень, остается актуальным.

Реальным претендентом на роль такой абстракции являются алгоритмически интерпретируемые расширения сетей Петри [1,5,6]. В настоящей работе предлагается воспользоваться технологией объектно-ориентированного программирования и принципов конкретизирующего программирования для развития базирующихся на интерпретируемых сетях методов построения модельных систем управления дискретными процессами в системах с императивным характером поведения. Объектом исследований являются задачи спецификации и интерпретации имитационных моделей, представленных системами формальных продукций в форме сетей элементарных переходов.

2. Сетевые модели дискретных процессов

В общем случае любые формальные модели описания дискретных процессов на определенном уровне абстрагирования могут рассматриваться как переходные системы. Потребность подобного рассмотрения диктуется последующим этапом конструирования вычислительной схемы на этапе программной реализации системы.

Переходная система S_p определяется тройкой

(1) $$S_p = \langle P_p, T_p, R_p \rangle,$$

где P_p – множество состояний, не обязательно конечное; T_p – множество переходов, причем конечное, что определяется требованием морфологической устойчивости системы; R_p – множество отношений на P_p, биективно соответствующих переходам T_p.

Конструктивные приемы определения переходной системы характеризуются способами задания отношений R_p, отображающих поведение элементов системы. Известные интерпретации таких отношений соответствуют описанию дискретных процессов, например, в терминах событий, процессов или работ. С учетом того, что функционирование системы управления заключается в регулярном фиксировании фактов изменения переменных состояния с последующим изменением значений функционально зависящих от них выходных данных,

то задачу формализации управления на организационно-технологическом уровне можно представить в терминах абстрактных СУД.

Однако формальное представление системы S_p набором продукционных правил не отражает динамики связи переменных ее состояния во времени. Задача интерпретации системы продукций – формирование упорядоченной последовательности действий, может решаться по схеме моделирования процессов на некоторой сетевой структуре. Сетевые структуры – элемент многих видов моделей, в частности, сетей Петри и их расширений [1,2].

Например, рассмотрим сеть Петри (рис.1), расширенную введением задержек в переходах – *TPN* (*Timed Petri Net*) [1,5]:

Сеть *TPN* – тривиальное расширение классических сетей Петри, определяется шестеркой

(2) $$TPN = \langle A, B, V, W, \mu, D \rangle,$$

где A и B – конечное множество элементов, называемых переходами и позициями, причем $A \cap B = \varnothing$ и $A \cup B \neq \varnothing$; V – функция инцидентности, отображающая связь между позициями и переходами $V = \langle A \times B \rangle \cup \langle B \times A \rangle \to \{0,1\}$; W – весовая функция, определенная на множестве положительных чисел N^+, $W = V \to N^+$; μ – разметка позиций сети, на множестве неотрицательных чисел N, $\mu : B \to N$; D – функция задержки переходов, определенная на множестве неотрицательных рациональных чисел R^+, $D : A \to R^+$.

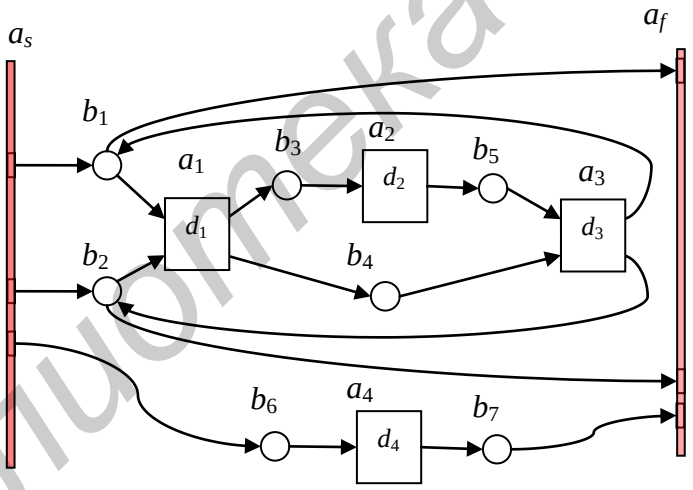

Рис. 1. Графическое представление временной сети Петри

Функционирование сети Петри любого вида заключается в переходе от разметки μ_k к разметке μ_{k+1}, вследствие срабатывания переходов. Здесь индекс k в обозначении вектора разметки $\mu_k = \{k, b \in B\}$ означает номер этапа функционирования *TPN*, связанного с моментом свершения особого события.

На траектории процесса функционирования *TPN* можно выделить особые события двух видов:

активизация перехода;

выход перехода из активного состояния или его пассивизация через интервал $d(a) \in D$, $a \in A$.

Обозначим для каждого элемента *TPN* $x \in A \cap B$ множество входных элементов $'x : 'x = \{y \mid y\, V\, x\}$, а множество выходных элементов – $x' : x' = \{y \mid x\, V\, y\}$.

Интерпретация процесса функционирования сети Петри в общем случае включает определение:

условий активизации переходов;

действий, реализуемых при активизации и пассивизации переходов.

Условие активизации переходов в *TPN* при некоторой разметке μ_k: переход $a \in A$ будет активизирован тогда и только тогда, когда истинно условие $\mu_k(b) \geq w(b,a)\ \forall b \in {}'a$, где $w(b,a)$ – вес связи позиции $b \in B$ и перехода $a \in A$.

Активизация перехода a в момент времени $S_k(a)$ на этапе k приведет к новой разметке μ_{k+1}, определяемой выражением:

$$(3)\qquad \begin{cases} \mu_{k+1}(b) = \mu_k(b) - w(b,a)\ \forall b \in {}'a; \\ \mu_{k+1}(b) = \mu_k(b)\ \forall b \in B \setminus {}'a. \end{cases}$$

Активизированный переход a после задержки во времени на интервале $d(a) \in D$ в момент времени $f_l(a) = S_k(a) + d(a)$ на этапе $l > k$ перейдет в пассивное состояние. Если в момент времени $f_l(a) - 0$, предшествующей пассивизации перехода a, разметка сети была μ_l, то пассивизация перехода приведет к новой разметке μ_{l+1}, определяемой выражением:

$$(4)\qquad \begin{cases} \mu_{l+1}(b) = \mu_l(b) + w(b,a)\ \forall b \in a'; \\ \mu_{l+1}(b) = \mu_l(b)\ \forall b \in B \setminus a'. \end{cases}$$

где $w(a,b) \in W$ – вес связи перехода $a \in A$ и позиции $b \in B$.

Если положить $d(a)=0$ для любых $a \in A$, то получим формальное определение классической сети Петри [1].

Учитывая инвариантность схемы проверок условий активизации переходов и их действий от структуры сети, конкретная моделируемая на основе *TPN* система может быть представлена структурой смежности, вектором задержек переходов и вектором разметки позиций. Именно такое представление оказывается достаточным для конструирования компоненты – элемента контура управления.

Объем памяти S_{TPN} для размещения статического описания *TPN* [5]:

$$(5)\qquad S_{\text{TPN}} = 4 + 5|A| + 2(|B| + \sum_{a \in A} |a'|) + 3\sum_{a \in A} |'a| .$$

В частности, структура смежности рассматриваемого варианта *TPN* (рис. 1) имеет вид:

a_s: b_1, b_2, b_6;
a_1: b_3, b_4;
a_2: b_5;
a_3: b_2, b_1;
a_4: b_7;
b_1: a_f, a_1;
b_2: a_f, a_1;
b_3: a_2;
b_4: a_3;
b_5: a_3;
b_6: a_4;
b_7: a_f.

Вектор задержек переходов $D = \{0, d_1, d_2, d_3, d_4, 0\}$.

Переменные состояния здесь (вектор разметки) образуют множество $\mu=\{b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$. При этом предполагается, что в исходный момент времени начальное состояние $\mu_0=\{0,0, 0, 0, 0, 0, 0\}$, а веса дуг *TPN* являются единичными. Удобство подобного предположения обусловлено простотой построения рекуррентной схемы моделирования процессов на сети с произвольной структурой [6].

Определение *TPN* в форме (2)-(4) соответствует ее структурной спецификации. Процессы на сети приходится рассматривать моделированием эволюции состояния элементов сети методами имитационного моделирования дискретных динамических систем. В теории сетей Петри иногда рассматривают отношения "сеть-система" и "сеть-процесс". Будем далее называть моделирование процессов на сети ее интерпретацией.

Итак, задача интерпретации *TPN* формулируется следующим образом:

задана сеть $TPN = \langle A, B, V, W, \mu, D \rangle$, необходимо построить последовательность $\langle f_i, i \rangle_k$, k=0,1,…, где k – номер этапа, когда в момент времени f_i изменено состояние перехода $i \in A$.

Для построения рекуррентной схемы моделировпания процессов на *TPN* удобно ввести понятия:

а) стартовый переход – дополнительный переход *s*, для которого справедливо

$$(6)\qquad \begin{cases} {}'s = \varnothing; \\ s' = \{b \mid \mu_0(b) > 0,\ b \in B\}; \\ w(s, x) = \mu_0(x),\ x \in s'; \\ d(s) = 0; \end{cases}$$

b) финишный переход – дополнительный переход *f***,** для которого справедливо

$$(7)\qquad \begin{cases} {}'f = \{b \mid \mu_n(b) > 0,\ b \in B\}; \\ f' = \varnothing; \\ w(x, f) = \mu_n(x),\ x \in {}'f; \\ d(f) = 0. \end{cases}$$

Сеть, дополненная переходами *s* и *f*, имеет нулевую начальную и конечную разметки, что упрощает ее объектно-ориентированное представление.

Изменение состояния сети можно однозначно привязать к моментам пассивизации переходов. Состояние сети в этом случае представляется списком событий

$$\begin{cases} L_0 = \langle 0, s \rangle, s \in A; \\ L_k = \langle t_i, a \rangle, i > k-1, a \in A, k > 0. \end{cases}$$

Пусть в некоторый момент времени $t=min\{t_i \in L_k\}$ зафиксирован факт пассивизации некоторого перехода $a \in A$. Обработка последствий подобных событий требует наличия следующих процедур:

$Step_1$ – моделирование действий перехода *a* при передаче меток в позиции $b \in a'$;

$Step_2$ – организация обработки последствий изменения разметки в позициях $b \in a'$ (последовательность просмотра таких позиций определяет способ устранения конфликтов);

$Step_3$ – проверка возможности активизации новых переходов из множества $\{x \in {}^{\backprime}y,\ y \in b,\ b \in a^{\backprime}\}$;

$Step_4$ – активизация перехода x, если необходимость этого установлена процедурой $Step_3$ (изменение разметки в позициях ${}^{\backprime}x$ и планирование нового события в момент $t+d(x)$);

$Step_5$ – фиксация нового особого события $(t+d(x), x)$;

$Step_6$ – выбор очередного события на *TPN* и повторение цикла моделирования, если список событий не пуст.

Алгоритм моделирования процессов на TPN в результате можно представить в следующем рекуррентном виде:

$$(8)\qquad \begin{cases} L_0 = \{\Phi, s\}, s \in A; \\ L_{k+1} = Step_6(Step_1, Step_2(Step_3(Step_4(Step_5(L_k)))), k > 0. \end{cases}$$

Очевидно, что *TPN* в форме (2)-(4) удобна для случая сетей с постоянными параметрами. Если какие-то параметры подобного определения зависят от времени, а также в случае существования некой структурной регулярности, то возникает естественное стремление формализовать такие зависимости функциями. Например, сеть Петри для задачи об обедающих мудрецах [1], обладает регулярной структурой и может быть задана функционально. Учитывая, что использование *TPN* в большинстве применений связано с необходимостью алгоритмической интерпретации, введение функций не является чем-то исключительным.

3. Спецификация процессов полиморфными сетями

Потребность использования иерархий абстрактных классов систем объектно-ориентированного программирования [2,8] обусловлена, например, тем, что отображение реальных процедур информационного обмена между объектами требует внесения следующих дополнений в сетевое описание:

- расширения понятия разметки позиций сети до множества значений переменных с различными формами внутреннего представления;
- определения условий активизации переходов в виде конъюнкции двуместных предикатов на множестве переменных модели;
- использования двух видов действий переходов сети – задержки во времени и операций над переменными модели;
- определения структур на множестве переменных (массивов с соответствующими средствами индексации элементов и агрегатов – поименованных упорядоченных совокупностей переменных);
- неявного динамического определения переходов, соответствующих фазам изменения состояний процесса приема-передачи сообщений, команд и директив оператора.

Рассмотрим схему спецификации дискретных процессов в рамках формальных СУД, обобщающую на определенном уровне абстрагирования известные схемы описания дискретных процессов. Далее обсудим вопросы перехода от формального представления переменных состояний к разнообразным объектам базовых типов данных системы программирования в терминах технологии объектно-ориентированного программирования (ООП).

По определению, СУД – тройка вида $S_c = \langle V_c, P_c, A_c \rangle$, где V_c – множество переменных состояния, имеющих область определения D_c; P_c – множество условий на V_c, задаваемых вычислимыми предикатами; A_c – множество действий, биективно соответствующих условиям P_c.

Каждое действие A_c – множество преобразований вида $V_c \Leftarrow A_c(V_c)$, причем правые части этих преобразований обязательно вычислимы на V_c, а все члены правой части подвергаются преобразованию одновременно.

Независимо от вида СУД, обсуждение вопросов управления дискретными процессами приводит к необходимости рассмотрения отношения вида "система - процесс". Анализ структуры процесса, порождаемого на любой СУД посредством регулярного применения процедур реализации элементарных действий из множества A_c, показывает, что множество преобразований вида $V_c \Leftarrow A_c(V_c)$ может рассматриваться как автономный процесс, определяемый относительно моментов изменения переменных правой части [1,5].

Любой элемент V_c^i из множества переменных состояния системы V_c может ассоциироваться с пятеркой функциональных зависимостей

(9) $$V_c^i = (E_i, C_i, D_i, F_i, I_i),$$

где E_i – условия активизации элемента, $E_i \in P_c$; C_i – действия, вызываемые в системе при входе элемента в активное состояние, $C_i \in A_c$; D_i – длительность во времени активной фазы элемента; F_i – действие перехода при выходе элемента из активного состояния, $F_i \in A_c$; I_i – действие перехода при прерывании его активного или пассивного состояния, $I_i \in A_c$.

Процесс функционирования переходной системы представляет собой регулярное применение процедур реализации элементарных действий из множества A_c. Однако, в отличие от *TPN*, порядок вызова процедур не очевиден.

Обозначим $\mathrm{dom}(f)$ – множество переменных состояния, связываемых функцией *f*. Интерпретация такого понятия – глобальные переменные модуля программы, реализующего функцию *f*. В терминах ООП, когда функцией *f* представлена функцией-элементом класса, $\mathrm{dom}(f)$ может включать элементы данных класса.

Множество переменных состояния отдельного перехода *i*

(10) $$V_i = \mathrm{dom}(E_i) \cup \mathrm{dom}(C_i) \cup \mathrm{dom}(F_i) \cup \mathrm{dom}(I_i).$$

Естественно предполагать, что порождение процесса активизации переходов основано на восприимчивости отдельных переходов к изменению только локальных переменных состояния. Последнее имеет как физическую аналогию, так и соответствует практической реализуемости функций перехода в вычислительных средах ЭВМ с архитектурой фон Неймана.

Отсюда следует, что на переходах СУД можно формально построить однодольную сеть

(11) $$IPN = (A, B, V),$$

связывающую функцией инцидентности *V* переходы *A* с потенциальной возможностью активизации, $V : (A \times A) \to \{0,1\}$. Переменные состояния здесь образуют множество *B*.

В случае *TPN*, позиции соответствуют переменным состояния, а переходы – действиям системы. Для рассматриваемого примера *TPN* (рис. 1) b_1, b_2, b_3, b_4, b_5, b_6, b_7 – переменные состояния; a_s, a_1, a_2, a_3, a_4, a_f – действия, для которых функции E_i, C_i, D_i и F_i очевидным образом следуют из определения *TPN*.

Однозначность поведения сети *IPN* во времени следует из возможности использования следующей регулярной схемы взаимодействия переходов. Если некоторый переход a_i стал пассивным, то это порождает необходимость проверки условий активизации переходов $a_{i+1} \in a_i'$, множества функций E_{i+1} которых содержат переменные, измененные функцией F_i. Если условие E_{i+1} истинно, то выполняются действия C_{i+1}, после чего активизируется a_{i+1} и проверяется возможность активизации переходов, условия активизации которых определены и на переменных функции C_{i+1}.

Таким образом, структура смежности сети переходов *IPN* формально определяется выражением

(12) $$a_x : a_x' = \{ a_x : a_x \mid \mathrm{dom}(E_a) \cap \mathrm{dom}(F_x) \neq \varnothing,\ a \in A \},\ a_x \in A \}.$$

Структура смежности связей переходов сети для рассматриваемого варианта *IPN* соответствует однодольному графу (рис.2):

$$\begin{cases} a_s : a_1, a_4; \\ a_1 : a_2, a_3; \\ a_2 : a_3; \\ a_3 : a_f, a_1; \\ a_4 : a_f. \end{cases}$$

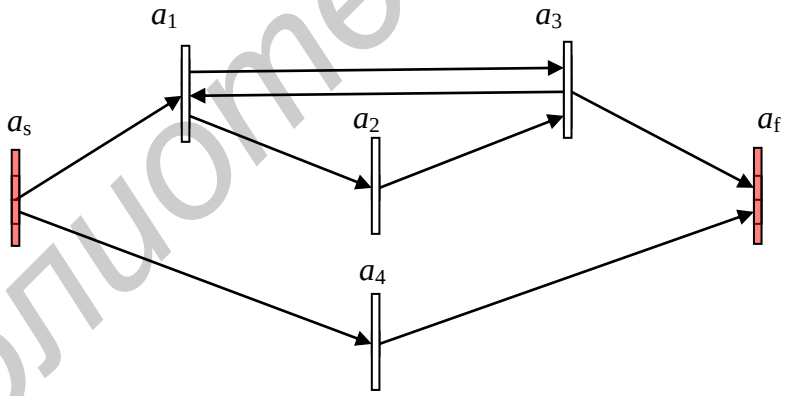

Рис. 2. Граф связей переходов сети *IPN*

Используя функциональный способ [5] определения процессов на *IPN*, память требуется лишь для списка событий и структуры смежности переходов

(13) $$S_{\mathrm{IPN}} = 1 + 3|A| + \sum_{a \in A} |a'|.$$

Для решения практических задач описание системы продукций конкретизируется в реальной вычислительной среде. Например, *TPN*, изображенная на рис.1, может быть закодирована средствами языка C++.

Функциональное описание перехода *IPN* как элемента сети пусть представлено в форме шаблона класса[2,4,8]:

```
template <class Time> struct Transition {
 int  E_next; // Ссылка на следующий активизированный переход
 Time E_time; // Момент пассивизации перехода
 virtual int  E(int) = 0;
 virtual void C(int) = 0;
 virtual Time D(int) = 0;
 virtual void F(int) = 0;
 virtual void I(int){}
};
```

Здесь функции-элементы класса [4,8] соответствуют по идентификаторам элементам пятерки V_c^i (9), где индекс *i* относится к экземпляру объекта – перехода некоторой сети, а целочисленный параметр функций предназначен для различения экземпляров сетей.

Параметр шаблона – class Time, определяет домен представления временных интервалов (пространство *R*+). Однако схема класса Transition, в отличие от версии, предложенной в [5], здесь содержит следующие отличия:

- включение элемента списка событий (E_next, E_time) в описание перехода позволяет исключить необходимость его декларации в области переменных состояния переходов;
- каждая функция перехода может, при необходимости, получать номер экземпляра сети, что упрощает моделирование однородных фрагментов сети.

Исходное описание *TPN* (рис. 1) тогда можно записать в виде:

```
// Схема идентификации классов переходов

#define E_(atom) _E_class##atom

// Схема нумерации идентификаторов

#define p(atom) _prefix_##atom

// ТЕКСТ ДЕТАЛИЗАЦИИ ОПИСАНИЯ МОДЕЛИ

#include <stdio.h>

long d1=2, d2=3, d3=4, d4=10;

int count;

// ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ СОСТОЯНИЯ

int b1, // 0
    b2, // 1
    b3, // 2
    b4, // 3
    b5, // 4
    b6, // 5
    b7; // 6

// ОПРЕДЕЛЕНИЕ БАЗОВЫХ КЛАССОВ ПЕРЕХОДОВ

template <class Time> struct E_(as): public Transition<Time> {
 virtual int   E(int) { return 0; }
 virtual void C(int) {
  b1=b2=b3=b4=b5=b6=b7=0;
  count=0;
  b1++,b2++,b6++;
```

```
 }
 virtual Time D(int) { return 0; }
 virtual void F(int) {}
};

template <class Time> struct E_(af): public Transition<Time> {
 virtual int  E(int) { return (b1 && b2 && b7); }
 virtual void C(int) {
  b1--, b2--, b7--;
}
 virtual Time D(int) { return 0; }
 virtual void F(int) {}
};

template <class Time> struct E_(a1): public Transition<Time> {
 virtual int  E(int) { return (b1 && b2); }
 virtual void C(int) { b1--, b2--; }
 virtual Time D(int) { return (d1); }
 virtual void F(int) { b3++, b4++; }
};

template <class Time> struct E_(a2): public Transition<Time> {
 virtual int  E(int) { return (b3); }
 virtual void C(int) { b3--; }
 virtual Time D(int) { return (d2); }
 virtual void F(int) { b5++; }
};

template <class Time> struct E_(a3): public Transition<Time> {
 virtual int  E(int) { return (b4 && b5); }
 virtual void C(int) { b4--, b5--; }
 virtual Time D(int) { return(d3); }
 virtual void F(int) { b1++, b2++; }
};

template <class Time> struct E_(a4): public Transition<Time> {
 virtual int  E(int) { return (b6); }
 virtual void C(int) { b6--; }
 virtual Time D(int) { return (d4); }
 virtual void F(int) { b7++; }
};
```

Представленный набор полиморфных классов допускает детализацию и уточнение в рамках стандартных возможностей языка C++[8].

Можно заметить, что формальное пространство переменных состояния здесь расширено глобальными переменными, используемыми функциями переходов. Таким образом, нет ограничений на использование стандартных лексических возможностей языков C/C++[8].

Множества переходов *IPN* для моделирования рассматриваемого варианта *TPN* (рис. 1) легко представить массивом

```
Transition<long> *L_func[] = { // Описание переходов сети
 new E_(as)<long>, // as
 new E_(a1)<long>, // a1
 new E_(a2)<long>, // a2
 new E_(a3)<long>, // a3
 new E_(a4)<long>, // a4
 new E_(af)<long>, // af
};
```

Описание сети связи переходов и позиций следует из определения *IPN*, но легко показать, что предпочтительная форма его представления – структура смежности в виде *FSF* (*Forward Star Form*)[5,9]:

```
int m_tran[] = { // Связи переходов
  /* 0 */ p(a1),p(a4), // as
  /* 2 */ p(a2),p(a3), // a1
  /* 4 */ p(a3),       // a2
  /* 5 */ p(af),p(a1), // a3
  /* 7 */ p(af),       // a4
  /* 8 */              // af
  /* 8 */
};

int l_tran[] = { // Индексы списка связанных переходов
  0,2,4,5,7,8,8
};
```

Таким образом, исходное описание моделируемой системы представлено непосредственно на языке программирования. Последнее допускает конкретизацию функций перехода в различных условиях применения модели.

4. Интерпретация полиморфных сетей

Задача интерпретации *IPN* формулируется следующим образом:

задана сеть $IPN = \langle A, B, V \rangle$, необходимо построить последовательность $\langle f_i, i_k \rangle$, k=0,1,…, где k – номер этапа, когда в момент времени f_i пассивизирован переход $i \in A$.

Для построения рекуррентной схемы моделирования процессов на сети *IPN*, подобно (6) и (7) для *TPN*, естественно ввести понятия:

a) стартовый переход – дополнительный переход s, для которого справедливо

$$(14)\qquad \begin{cases} s' = \varnothing, \\ s' = \{a \mid \mathrm{dom}\langle E_a \rangle \cap \mathrm{dom}\langle F_s \rangle \neq \varnothing,\ a \in A\}, \\ d\langle \rangle = 0; \end{cases}$$

b) финишный переход – дополнительный переход ***f***, для которого

$$(15)\qquad \begin{cases} f' = \varnothing, \\ d\langle f \rangle = 0. \end{cases}$$

Сеть *IPN*, дополненная переходами s и f, будет называться стандартной. Далее будет показано, что стартовый и финишный переходы соответствуют понятиям конструктора и деструктора класса, представляющего переходы сети.

Обработка последствий пассивизации любого перехода $a \in A$ требует наличия следующих процедур:

$Step_1$ – активизация модуля выходных действий перехода a;

$Step_2$ – организация обработки последствий локального изменения переменных состояния;

$Step_3$ – проверка возможности активизации новых переходов из множества $\{x \in I'\}$;

$Step_4$ – активизация перехода x, если необходимость этого установлена процедурой $Step_3$ (активизация перехода x влечет вызов функции входных действий и планирование нового события в момент $t+d(x)$);

$Step_5$ – фиксация нового особого события, представляемого кортежом $(t+d(x), x)$, в списке событий;

$Step_6$ – выбор очередного события на *IPN* и повторение цикла моделирования, если список событий не пуст.

Таким образом, с точностью до содержания отдельных функций, рекуррентная схема процесса пассивизации переходов сетей *IPN* совпадает с ранее рассмотренной схемой (8) для *TPN*:

$$(16) \qquad \begin{cases} L_0 = \langle \emptyset, s \rangle, s \in A; \\ L_{k+1} = Step_6 \langle Step_1, Step_2 \langle Step_3 \langle Step_4 \langle Step_5 \langle L_k \rangle \rangle \rangle \rangle \rangle, \; k > 0. \end{cases}$$

Предварительно выделим следующие классы, инвариантные относительно вида сети, но требующиеся для построения отображения “сеть-процесс”:

LinkIterator – итератор списка смежных дуг;

EventScaner – монитор управления последовательностью событий.

Итератор списка смежных дуг предназначен для выборки описания связей элементов сети, требующихся при обработке особых событий.

Предполагая представление структуры сети в форме структур смежности, предложенной в [7], итератор можно записать в виде:

```
class LinkIterator {
  int  base;
  int  stop;
  int  temp;
public:
  LinkIterator(int node, int *link):
   temp(link[node]), stop(link[node+1]) { base = temp; }

// Выборка очередной связи
  bool operator()(int &item) {
   if (temp<stop) {
    item = temp++;
    return true;
   }
   return false;
  }

// Восстановление итератора
  void reset() { temp = base; }
};
```

Монитор управления последовательностью событий обеспечивает синхронизацию элементарных процессов на сети. Представляемый далее шаблон класса EventScaner предназначен для представления списка событий на множестве пассивизируемых переходов сети

```
// Базовый класс интерпретации переходных процессов на сети

template <class Time> class EventScaner {
 int n_tran;
```

```
protected:

 Transition<Time> **F_tran;
 int index;
 virtual void start() {}
 virtual void prolog() = 0;
 virtual void finished(int t) = 0;
 virtual void future(int n, Time f) {}
 virtual void fired(int n, Time f) {}
 virtual void stop() {}

public:

 EventScaner(int Nxtran, Transition<Time> **Fxtran,int Index):
  n_tran(Nxtran),F_tran(Fxtran),index(Index) {}

 void operator()() {

  // Фиксация позиции стартового перехода

  int  &head = F_tran[0]->E_next;
  Time &temp = F_tran[0]->E_time;

  // Инициализация списка событий
  head = 0; // Стартовый переход имеет нулевой номер
  temp = 0; // Отсчет относительного времени от нуля
  start(index,temp);

  // Фиксация пассивности переходов
  for (int item=1; item<n_tran; F_tran[item++]->E_next=n_tran);

  prolog();

  // Обработка последовательности событий
  do {
   item = head; // Выборка очередного перехода
   head = F_tran[item]->E_next;
   temp = F_tran[item]->E_time; // Момент пассивизации
   if (item) F_tran[item]->E_next=n_tran;
   fired(item,temp);
   finished(item);
  } while (head);
  stop(index,temp);
 }

 void planer(int n, Time delta) {
  if (F_tran[n]->E_next==n_tran) {
   int  &head = F_tran[0]->E_next;
   Time &temp = F_tran[0]->E_time;
   Time f=F_tran[n]->E_time=temp + delta;
   future(n,f);
   for (int p=0, q=head;
        (q>0)&&(f>=F_tran[q]->E_time); p=q, q=F_tran[p]->E_next);
   F_tran[n]->E_next=q, F_tran[p]->E_next=n;
  }
 }
 ~EventScaner() {}
};
```

Отметим, что побочным результатом явного выделения монитора событий является инкапсуляция множества собственных событий моделируемого элемента системы. Конструктор класса EventScaner формирует область определения потенциально возможных событий по априорно заданному количеству переходов сети.

Виртуальные функции класса EventScaner введены для образования интерфейса связи пользователя класса с процессом моделирования на различных этапах развертки выражения (16).

Динамическое описание функционирования сети *IPN* представлено следующим классом:

```
// Интерпретатор однодольной сети переходов

template <class Time>
 class SingleStepProcess: public EventScaner<Time> {
  virtual void prolog(); // Входные действия
  virtual void expanded(int item);

  protected:

  int *L_tran;
  int *M_tran;
  virtual void expans(int n); // Попытка активизации перехода n

  public:

   SingleStepProcess(
    Transition<Time> **T, int N, int *L, int *M):
    EventScaner<Time>(N,T), L_tran(L), M_tran(M) { }

   // Обработка последствий пассивизации перехода item
   void finished(int item);

   ~SingleStepProcess() {}
  };
```

Класс SingleStepProcess наследует монитор управления последовательностью событий EventScaner и параметризован относительно типа данных представления времени (class Time). Рассмотрим далее его функции-элементы.

Пролог процесса интерпретации однодольной сети переходов обеспечивает поддержку понятия стандартного вида интерпретируемой сети:

```
template <class Time>
 void SingleStepProcess<Time>::prolog() {
  F_tran[0]->C(index); // Входные действия
 }
```

Обработка последствий пассивизации перехода item выполняется виртуальной функцией

```
template <class Time>
 void SingleStepProcess<Time>::expanded(int item) {
  expans(item);
 }
```

(ее введение позволит далее обеспечить возможность расширения конструкции сети, например, посредством определения вложенных переходов).

Попытка активизации перехода item моделируется виртуальной функцией

```
template <class Time>
void SingleStepProcess<Time>::expans(int item) {
 Transition<Time> *f_tran = F_tran[n];
 if (f_tran->E(index)) { // Проверка условий активизации
  f_tran->C(index); // Входные действия
  // Планирование момента пассивизации перехода
  planer(n,f_tran->D(index));
 }
}
```

Обработка последствий пассивизации перехода item

```
template <class Time>
 void SingleStepProcess<Time>::finished(int item) {
  F_tran[item]->F(index); // Выходные действия
  ListIterator ExistTran(item, L_tran, M_tran);
  while (ExistTran(item)) expanded(item);
 }
```

Следует отметить, что переходы *IPN* в рассмотренном варианте класса интерпретатора связаны относительно моментов пассивизации. Фиксация такого правила интерпретации процессов на сети позволяет достичь однозначности эволюции ее состояния.

Таким образом, определение *IPN* не использует понятия позиции или ее разметки. Операции при проверке условий, активизации и пассивизации переходов ассоциируются с переходом отдельными функциями. Представленный здесь параметризованный класс SingleStepProcess не имеет ограничений на конкретизацию смысла переходной системы на уровне языка C++ (пример применения для моделирования представленной рис.1 *TPN* размещен в приложении).

5. Заключение

Принцип восприимчивости системы к изменению локальных переменных состояния, лежащий в основе сетей Петри, оказывается привлекательным не только для формального моделирования процессов, но и для конструктивного применения в качестве технологической основы программной реализации системы.

Показано, что описание дискретных процессов может быть определено сетями переходов, каждый из которых представлен полиморфными классами в терминах объектно-ориентированных технологий. Это позволяют определить практически ничем не ограниченное понятие расширения сети Петри.

Полиморфные сетевые описания могут рассматриваться как спецификации имитационных моделей и законов логического управления, которые допускают детализацию и конкретизацию на условия применения.

Построенная иерархия базовых классов предоставляет интерфейсы для эволюционного определения или уточнения как функций перехода сети, так и действий на отдельных фазах моделирования процессов на сети. Тем самым,

обеспечивается возможность использования как известных вариантов сетей Петри, так и их расширений в рамках операционных возможностей языка С++.

Вне рассмотрения в работе оставлены вопросы моделирования вложенных сетевых моделей, процедуры проверки формальных свойств сетей Петри, например, живость и достижимость. Однако очевидно, что известные методы их разрешения реализуются расширением предложенных классов.

Приложение. Пример программы моделирования

```
//  Моделирования процессов на TPN, изображенной на рис.1

#define Id(atom) {#atom}

char *Idents[] = {
 Id(as),Id(a1),Id(a2),Id(a3),Id(a4),Id(af)
};

template <class Time>
class Single: public SingleStepProcess<Time> {
public:
 Single(Transition<Time> **F, int N, int *L, int *M, int I=0):
  SingleStepProcess<Time>(F,N,L,M,I) {}
 virtual void start(int i, Time f) {
  printf("\nFired         Future");
 }
 virtual void future(int n, Time f) {
  printf("\n              %s: %3d", Idents[n],f);
 }
 virtual void fired(int n, Time f) {
  printf("\n%s: %3d", Idents[n],f);
 }
 virtual void stop(int i, Time f) {
  printf("\nFinal at %d",f);
 }
};

void main() {
 Single<long>
  Obj(L_func,sizeof(L_func)/sizeof(*L_func),l_tran, m_tran);
 Obj();
}
```

Результаты работы программы:

```
Fired         Future
as:   0
              a1:   2
              a4:  10
a1:   2
              a2:   5
a2:   5
              a3:   9
a3:   9
              a1:  11
a4:  10
a1:  11
              a2:  14
```

```
a2:  14
                a3:  18
a3:  18
                af:  18
af:  18
Final at 18
```

Список литературы

1. Питерсон Дж. Теория сетей Петри и моделирование систем/Пер с англ. – М.: Мир, 1984. – 284 с.
2. Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования.– М.: Мир, 1999. – 191 с.
3. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации. – СПб.: Наука, 2000. – 780 с.
4. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно–ориентированного проектирования. Паттерны проектирования/Пер. с англ. – СПб.: Питер, 2001. – 386 с.
5. Ревотюк М.П., Тихомирова Е.В. Алгоритмическая интерпретация процессов на временных сетях Петри //Вестник Брестского государственного технического университета. – 2001.– № 5(11).– С. 86-90.
6. Ревотюк М.П., Тихомирова Е.В. Базовый класс интерпретатора процессов на расширенных временных сетях Петри // Моделирование и информационные технологии проектирования: Сб. науч. тр. /Мн.: ИТК НАН Беларуси, 2001. – С. 45-56.
7. Шеннон Р. Имитационное моделирование систем – искусство и наука/Пер. с англ. – М.: Мир, 1978. – 418 с.
8. Страуструп Б. Язык программирования С++. – М.: БИНОМ, 2001. – 1099 с.
9. Седжвик Р. Фундаментальные алгоритмы на С++. Части 1-4. Анализ. Структуры данных. Сортировка. Поиск – Киев: ДиаСофт, 2002. – 687 с.