AUTOMATYKA • 2007 • Tom 11 • Zeszyt 1–2

Wojciech Szmuc*

Modelowanie konstrukcji obiektowych języka UML z zastosowaniem kolorowanych sieci Petriego

1. Wprowadzenie

Modelowanie obiektowe jest obecnie głównym podejściem do wytwarzania oprogramowania złożonych systemów oprogramowania. Podejście to ma wiele zalet, skutkujących jego powszechnym zastosowaniem, nie zapewnia jednak wystarczających mechanizmów weryfikacji właściwości tworzonego systemu. Powoduje to konieczność zastosowania dodatkowych narzędzi pozwalających na pełniejszą analizę właściwości systemu. Istniejące metody formalne stosowane aktualnie do opisu oraz analizy prostych systemów, nie doczekały się istotnego włączenia w proces wytwarzania oprogramowania. Szczególnie ważne byłoby zastosowanie tych metod w technice obiektowej, gdyż umożliwiłoby to stworzenie równoległej ścieżki formalnego modelowania oraz weryfikacji, uzupełniających klasyczny (inżynierski) wątek rozwijania oprogramowania. Wspomniane rozszerzenie procesu wytwarzania umożliwiłoby systematyczną analizę poprawności, co prowadziłoby do wcześniejszego wykrycia i usunięcia błędów.

Jednym z powszechniej stosowanych języków opisu systemów w technologii obiektowej jest UML (*Unified Modeling Language*) [1]. Język ten dostarcza środków do modelowania różnych perspektyw systemu, będąc przez to uniwersalnym narzędziem do opisu bardzo szerokiej klasy systemów w różnych fazach wytwarzania. Brak precyzyjnej (formalnej) definicji zarówno w obszarze składni, jak i semantyki uniemożliwia formalną analizę (dowodzenie) właściwości modeli opisanych z zastosowaniem UML. W badaniach korzystano z narzędzia TAU 2.4 [2] implementującego (nie w pełni) standard UML 2.0 [3].

Sieci Petriego [4] są z kolei metodą formalną, która wydaje się najpełniej modelować różne aspekty złożonych systemów. Daje to możliwość sprawdzenia właściwości pełnego opisu systemu, co w innych metodach formalnych jest często bardzo utrudnione ze względu na złożoność. Opracowany algorytm tworzy sieci realizowalne w CPN Design 4.0 (*Coloured Petri Nets Design*) [5].

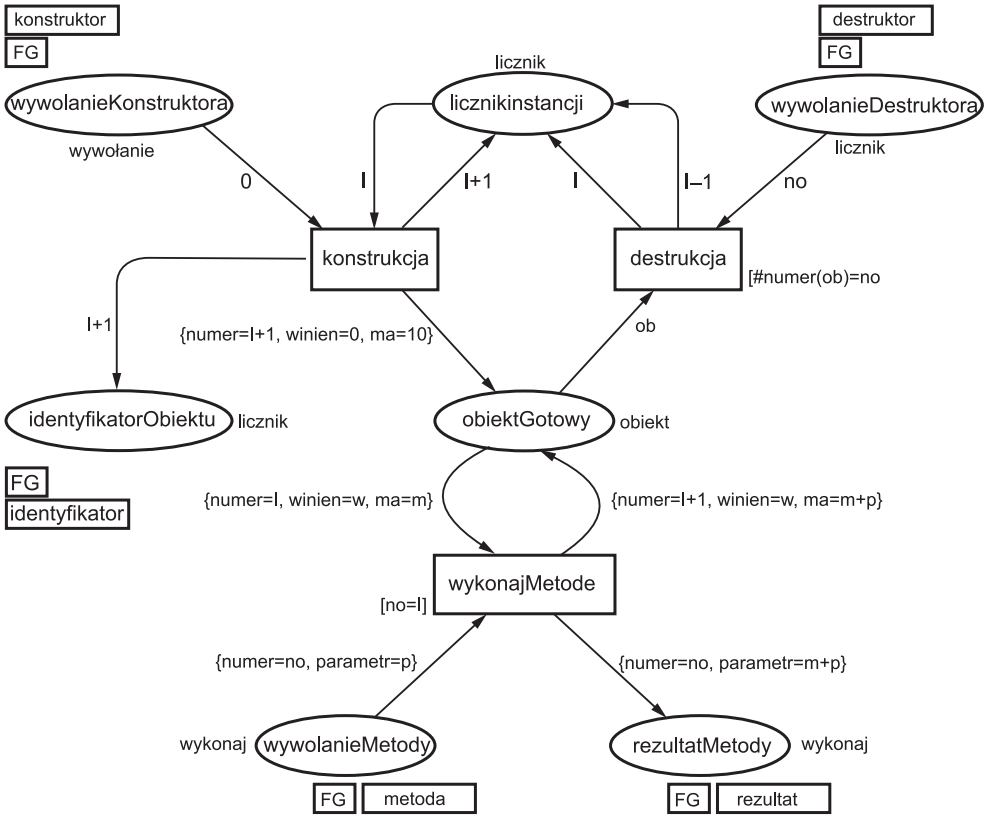
2. Modelowanie diagramów klas

Jednym z ważniejszych etapów projektowania oprogramowania obiektowego jest opisanie rodzajów obiektów, które są częściami systemu. UML definiuje osobny typ diagramu w celu przedstawienia widoku systemu od strony konstrukcji obiektowych.

* Katedra Telekomunikacji, Akademia Górniczo-Hutnicza w Krakowie

Diagram klas (*class diagram*) opisuje architekturę systemu – strukturę klas (atrybuty, metody), powiązania między typami (klasami, interfejsami, typami danych, itp.). Diagram klas nie jest reprezentowany w sieci Petriego, w formie bezpośredniej analogii. Wynika to z konieczności zinterpretowania mechanizmów obiektowych. Wprawdzie zostaje utworzona sieć zawierająca analogiczne informacje, ale ich forma może znacznie odbiegać od prostej analogii do diagramu klas.

Ogólnie strona przedstawiająca klasę zawiera sieci związane z realizacją konstruktora metod oraz destruktora (rys. 1). Sieci te mogą być (dla poprawienia czytelności) zrealizowane na osobnych stronach podstawionych pod przejścia.



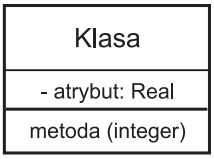
Rys. 1. Sieć Petriego reprezentująca klasę

Konstruktor tworzy nowy znacznik symbolizujący obiekt. Znacznik posiada pola związane z atrybutami oraz numerem obiektu. Ich inicjalizacja następuje w konstruktorze. Zwraca on znacznik obiektu, który jest jego numerem na stronie klasy. „Gotowy" obiekt jest umieszczany w miejscu, w którym oczekuje na wywołanie metody/destruktora (klasa pasywna), albo startuje maszynę stanową (klasa aktywna). Na stronie reprezentującej klasę znajduje się też licznik instancji. Jest on reprezentowany przez miejsce zawierające znacz-

nik typu całkowitego (**int**). Na początku ma wartość 0, lecz w miarę tworzenia kolejnych instancji jest ona zwiększana. Konstruktor obiektu pobiera ten znacznik a zwracając zwiększa jego wartość o 1. Destruktor natomiast zmniejsza wartość o 1.

Pierwszym krokiem konwersji jest utworzenie strony, na której mogą znaleźć się przedstawione niżej konstrukcje z UML.

Klasa (*class*) – abstrakcja grupy obiektów (rys. 2), które mają takie same właściwości (atrybuty), zachowania (metody) strukturę i relacje. Klasę można powołać do życia, tworząc obiekty (chyba, że jest to klasa abstrakcyjna), które mają takie same właściwości. Ogólnie klasa nie posiada swojego własnego sterowania (jest pasywna) w odróżnieniu od klasy aktywnej posiadającej przynajmniej jeden własny wątek.



Rys. 2. Symbol klasy

W sieci Petriego definicję klasy przedstawia się na osobnej stronie. Jest ona reprezentowana przez przejście w stronie nadrzędnej („diagram klas"), jeżeli nie jest częścią innej klasy.

Klasa abstrakcyjna (*abstract class*) jest klasą, która nie tworzy obiektów. Jest elementem ogólnym, który nie może być bezpośrednio powołany do życia. Kreacja może nastąpić dopiero po uszczegółowieniu (konkretyzacji). Nie jest to więc element, który wiąże bezpośrednio definicję z przebiegiem sterowania. Ma to swoje konsekwencje przy konstruowaniu sieci Petriego, gdyż podsieć bezpośrednio odpowiadająca elementowi jest po zakończeniu całej konwersji usuwana z sieci. Pierwszym krokiem konwersji jest przetłumaczenie elementu ogólnego na sieć Petriego. Następnie sieć jest wykorzystywana przy tworzeniu elementów specjalizowanych (*specialized*) w sposób analogiczny do stosowanego dla elementów wirtualnych (opisanych dalej). Po wygenerowaniu wszystkich specjalizacji usuwa się z modelu sieć bazową.

Klasa może posiadać parametry kontekstowe, które są precyzowane przez dziedziczącą ją klasę.

Wirtualność określa, czy klasa może zostać przedefiniowana. Ma to zastosowanie tylko wtedy, gdy klasa zawiera inną podklasę. W takim przypadku sieć Petriego jest tworzona według algorytmu dla relacji uogólnienia.

Uogólnienie (*generalization*) jest relacją pomiędzy dwiema sygnaturami (na przykład klasami albo metodami) wskazującą, że jedna z nich jest bardziej ogólną sygnaturą a druga jest bardziej szczegółową. Ta ostatnia dziedziczy wszystkie właściwości od bardziej ogólnej i może również posiadać dodatkowe właściwości. Z tego powodu relacja uogólnienia jest również znana jako relacja dziedziczenia. Jeżeli uogólnienie zostaje ustalone pomiędzy dwoma typami (na przykład dwiema klasami) bardziej szczegółowy typ określa podtyp bardziej ogólnego typu (który jest czasami nazywany nadtypem). Oznacza to, że instancja bar-

dziej ogólnego typu może zostać zastąpiona przez instancję typu bardziej szczegółowego. Uszczegółowienie może się wiązać z ustawieniem parametrów kontekstowych. W sieci Petriego relacja ta jest realizowana przez skopiowanie sieci typu bardziej ogólnego oraz dodaniu elementów uszczegóławiających. Jeżeli występują parametry kontekstowe, to ich typy określane są w miejscu wywołania. Na tej podstawie budowana jest sieć.

Każda klasa może posiadać atrybuty oraz metody. Modelowanie poszczególnych elementów opisu klasy w terminologii sieci Petriego jest opisane poniżej.

Atrybut (*attribute*) jest cechą, która może przechowywać jeden lub wiele wartości podczas pracy systemu. W sieci Petriego atrybuty proste są polami znacznika reprezentującego obiekt (powoływanego przez konstruktora) – każdemu atrybutowi odpowiada osobne pole znacznika.

Jeżeli atrybut jest typu klasa to jego wartość jest obiektem. W takim przypadku atrybut może być powiązany przez:

- powiązanie,
- agregacja – słowo kluczowe **shared**,
- kompozycja – słowo kluczowe **part**.

Powiązanie (*association*) jest relacją semantyczną pomiędzy dwoma lub wieloma klasyfikatorami, wskazującą, że instancje tych klasyfikatorów będą powiązane.

W sieci Petriego relacja ta jest reprezentowana przez łuki związane z korzystaniem przez instancję danej klasy, z instancji innej klasy. Jeżeli powiązanie nie jest skierowane (nie rozróżnia instancji korzystającej oraz wykorzystywanej), to jego uwzględnienie w sieci zapewnia tłumaczenie innych elementów UML.

Agregacja (*aggregation*) jest szczególnym rodzajem powiązania. Łączy ono instancję klasyfikatora agregującego znajdującą się na jednym z jej końców, z instancją klasyfikatora części znajdującą się na drugim końcu.

Część połączenia może występować w więcej niż jednej relacji tego typu.

Linia połączenia określa, że instancja klasy łączonej jest rozważana informacyjnie jako posiadana przez instancję klasy komponentu.

Sieć Petriego dla tej relacji jest taka sama jak dla relacji powiązania skierowanego.

Kompozycja (*composition*) jest szczególnym rodzajem agregacji. W kompozycji część jest posiadana przez relację i może przez to występować tylko w jednej relacji tego typu.

Linia kompozycji określa silną postać agregacji (związek całość/część), gdzie instancja klasy zawieranej istnieje tylko tak długo, jak długo istnieje instancja klasy zawierającej. W sieci Petriego konstruktor klasy zawierającej wywołuje konstruktora klasy zawieranej (analogicznie destruktory). Dla czytelności sieci klas będących częściami powinny znajdować się na stronie klasy zawierającej (jako diagram struktury złożonej).

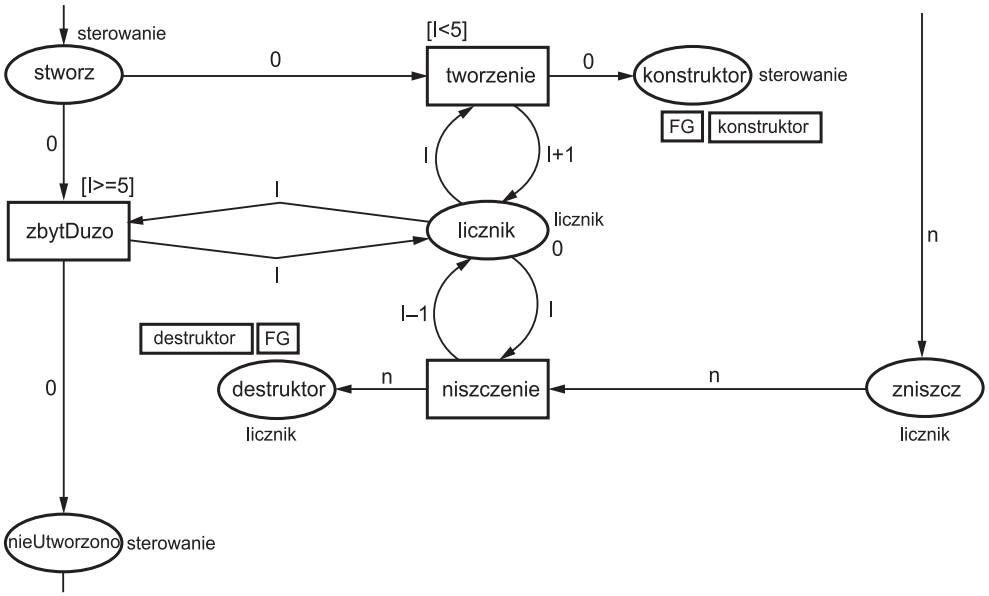
Jeżeli atrybut dostępny jest z zewnątrz, to w celu wykonania na nim operacji pobierany jest znacznik zawierający atrybuty.

UML wiąże tryb dostępu do atrybutów bardziej z pojęciem obiektu niż klasy (inaczej niż np: C++, Java). Konsekwencją tego jest brak dostępu metody obiektu do prywatnych pól innego obiektu nawet, jeżeli są to obiekty tej samej klasy.

Atrybuty mogą występować w liczebności większej niż 1 zapisywanej, w postaci: nazwa**:** typ**[**liczebność**]**. Liczebność (podczas pracy systemu) może się zawierać w granicach określonych przez wyrażenie, którego składnia jest następująca: liczebnośćMinimalna**..**liczebnośćMaksymalna. Gwiazdka (*****) oznacza brak ograniczeń.

W zależności od tego, czy liczebność atrybutu jest większa od 1, czy nie, faktyczny jego typ jest różny. Jeżeli liczebność jest większa od 1, to atrybut będzie typu kontener, który może przechowywać listę wartości. Jeżeli liczebność wynosi dokładnie 1 (lub 0**..**1), to nie stosuje się kontenera.

W zależności od tego, jakie biblioteki typów danych są dostępne, typ kontenera może być różny. Zazwyczaj poszczególne generatory kodu będą dostarczać różnych typów kontenerów dla zapewnienia odpowiedniej integracji z językiem docelowym. Jeżeli żadna specyficzna biblioteka nie została załadowana, to użyty zostanie typ **String**. W sieci Petriego konstrukcja związana z liczebnością zależy od użytego typu kontenera. Brak możliwości zdefiniowania tablicy sugeruje wykorzystanie listy.



Rys. 3. Sieć Petriego reprezentująca licznik instancji

Dostęp do tablicy może być uwarunkowany ograniczeniami wynikającymi z definicji atrybutu. Należy więc zastosować konstrukcję, która zezwala na powołanie/skasowanie instancji, jeżeli spełnione są warunki (rys. 3). W sieci reprezentującej kontener (w klasie zawierającej) znajduje się licznik instancji (obiektów). Jego działanie polega na zliczaniu wywołań konstruktora oraz destruktora. Wywołanie uzależnione jest od bieżącej liczby instancji oraz jej ograniczeń. Konstrukcja składa się z miejsca oraz trzech przejść. W miejscu znajduje się znacznik, w którym zapisana jest liczba instancji. Jedno z przejść odpowiada

za zliczanie wywołań konstruktora. Przejście to pobiera znacznik z miejsca i jeżeli jego wartość spełnia warunki utworzenia instancji, to jest on inkrementowany. Równocześnie zostaje wysłany znacznik do konstruktora obiektu. W odpowiedzi otrzymuje się znacznik zawierający numer utworzonego obiektu. Jest on zapisywany w kontenerze. Jeżeli obiekt nie może być utworzony, to inne przejście przesyła znacznik do miejsca wskazującego na niepowodzenie utworzenia kolejnej instancji. Trzecie z przejść odpowiada za wywołanie destruktora. Dekrementuje ono wartość znacznika instancji i przesyła otrzymany znacznik (z numerem obiektu) do destruktora.

Atrybut wyprowadzany (*derived*) – atrybut, którego wartość nie jest przechowywana, lecz obliczana w miejscu odwołania się do niej.

/piPół**: Real**

W sieci Petriego odpowiada to konstrukcji, w której algorytm jest zawarty na stronie podstawionej pod przejście. Przejście to jest używane (wysyłany jest do niego znacznik), gdy potrzebna jest wartość elementu wyprowadzanego. Wynik jest zwracany przez znacznik wychodzący z przejścia. Taka konstrukcja zapewnia jednolitość algorytmu dla wszystkich wywołań.

Atrybut statyczny (*static*) – jest atrybutem który, jest związany raczej z klasą niż instancją. Oznacza to, że istnieje tylko jedna instancja atrybutu współdzielona przez wszystkie instancje danej klasy.

static łańcuch**: String**

W sieci Petriego nie ma bezpośredniej możliwości przechowywania wartości zmiennej globalnej. Problem ten można rozwiązać przez utworzenie miejsca połączonego fuzją między instancjami, w którym będzie znajdować się znacznik przechowujący wartość atrybutu statycznego. Jeżeli pewna instancja będzie potrzebować jego wartości, to pobierze znacznik, przetworzy jego wartość i odda znacznik z powrotem (zasób współdzielony). Jeżeli atrybut jest obiektem, to zasobem współdzielonym jest znacznik (z numerem) tego obiektu.

Atrybut stały (*constant*) – atrybut, którego wartości nie można zmienić (wartość domyślna).

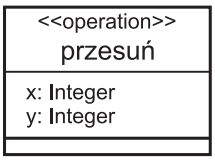
const Real pi**=**3.1415

W sieci Petriego deklaracja bez nazwy typu np.:

val pi**=**3.1415**;**

Należy jednak zwrócić uwagę, aby nie występowały stałe o tych samych nazwach, gdyż jest to deklaracja globalna.

Operacja (*operation*) jest deklaracją, wskazującą że instancje klasy będą w stanie przechwycić wywołania, które są zgodne sygnaturą metody (rys. 4). Metoda (implementacja operacji) jest wykonywana, jeżeli zostanie wywołana. To oznacza, że jeżeli odbiorca jest instancją pasywną, to implementacja zostanie wykonana natychmiast po wywołaniu metody. Jeżeli odbiorca jest instancją aktywną wykonanie implementacji może zostać opóźnione do pewnego momentu w przyszłości, gdy instancja będzie w stanie, w którym wywołanie metody jest akceptowane.

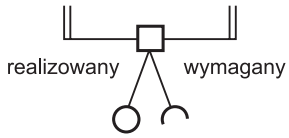


Rys. 4. Symbol operacji

W sieci Petriego metody nie są wprost deklarowane. Ich definicje znajdują się na stronie zawierającej definicję klasy. Możliwe jest automatyczne wytworzenie takiej strony przez konwersję [6] algorytmu metody zapisanego przez odpowiedni diagram stanów (*statechart diagram*).

Metoda jest wywoływana przez umieszczenie znacznika w miejscu wejściowym przejścia. Wykonanie uzależniane jest od obecności znacznika obiektu z takim samym numerem obiektu w miejscu oczekiwania (przejście ma 2 łuki wejściowe) – rysunek 1. W klasie aktywnej (opis dalej) znacznik obiektu jest przesyłany do metody w wyniku wykonywania się maszyny stanowej. Parametry dla metody przesyłane są przez pola znacznika wywołania metody.

Port (*port*) jest nazwanym punktem komunikacji (*interaction point*) klasy aktywnej. Określa on zrealizowany (przez klasę) interfejs oraz wymagania dotyczące interfejsów innych klas (rys. 5). Jest to statyczny (określony z góry) związek klasy aktywnej z otoczeniem niezależnie od tego czy instancja jest tworzona/niszczona statycznie czy dynamicznie. Porty mogą grupować zbiory interfejsów, które są udostępniane różnym udziałowcom.



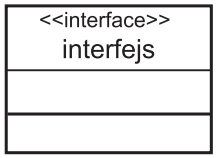
Rys. 5. Symbol portu oraz interfejsów

W sieci Petriego port jest reprezentowany przez miejsce pośredniczące w komunikacji, przez które przesyłane są sygnały. Jego kolor jest kolorem (typem) znacznika odpowiadającego sygnałowi. Jeżeli port ma przesyłać wiele różnych sygnałów, to w sieci Petriego należy zdefiniować jego kolor tak jak dla listy sygnałów.

W przypadku relacji uogólnienia między klasami, jeżeli występują porty należące do klasy nadrzędnej, to zostaną one odziedziczone.

Dla każdego portu można określić interfejs realizowany oraz wymagany. Realizowany interfejs (*realized interface*) portu określa przychodzące żądania, które mogą zostać obsłużone przez ten port. Wymagany (*required interface*) interfejs określa wychodzące żądania, które muszą zostać obsłużone przez klasę podłączoną do portu z zewnątrz przez jeden lub więcej łączników.

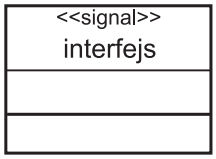
Interfejs (*interface*) jest klasyfikatorem, który porządkuje różne elementy komunikacji (nie jest osobnym obiektem). Określa on zbiór atrybutów, metod, oraz sygnałów, które muszą zostać zaimplementowane w klasie realizującej interfejs (rys. 6).



Rys. 6. Symbol interfejsu

Podczas konstrukcji sieci Petriego można go wykorzystać do deklaracji kolorów miejsc portowych – kolor musi uwzględniać wszystkie typy sygnałów, które mogą być przesłane przez port.

Sygnał (*signal*) jest jednym z podstawowych środków komunikacji w UML (rys. 7). Sygnał jest asynchronicznym komunikatem, który jest wysyłany pomiędzy klasami aktywnymi. Może on przekazywać dane, które muszą być zgodne z zadeklarowanym typem parametru sygnału.



Rys. 7. Symbol sygnału

W sieci Petriego sygnał jest reprezentowany przez znacznik, którego pola są parametrami sygnału. Znacznik jest zadeklarowany jako rekord, a typy jego pól odpowiadają typom parametrów sygnału. Jedno z pól przechowuje nazwę sygnału. Adres jest zapisywany w znaczniku w postaci rekordu zawierającego dwa pola. W pierwszym zapisywana jest nazwa przejścia, pod które podstawiana jest instancja strony z obiektem docelowym. Jeżeli strona nie jest podstawiana, to pierwsze pole adresu zawiera jej nazwę. Pole to jest typu łańcuchowego (**string**). Wartość „~” oznacza brak nazwy. W drugim polu przechowywany jest numer obiektu docelowego. Pole to jest typu całkowitego (**int**). Wartość ~1 oznacza brak numeru obiektu. Adres odbiorcy znajduje się na pierwszym polu (zawierającym rekord) znacznika sygnału. Adres nadawcy (analogiczny) znajduje się w drugim polu znacznika. Aby adresowanie działało poprawnie, wymagane są dodatkowe konstrukcje związane z przesyłaniem sygnałów do sieci docelowych. Jednym ze sposobów jest odpowiednie ustawienie dozorów przejść reprezentujących łączniki. Zezwalałyby one na przesłanie jedynie znaczników adresowanych do danej podsieci oraz jej podstron (również z adresem ~1). Takie adresowanie ma sens tylko w przypadku istnienia wielu odbiorców mogących odebrać sygnał.

Lista sygnałów (*signallist*) – grupuje związane ze sobą sygnały. Konstrukcja ta jest używana dla zwiększenia zwięzłości. Jest ona zazwyczaj wykorzystywana w portach oraz łącznikach, gdy wygodniej jest użyć listy niż wymieniać po kolei sygnały. Konstrukcja ta nie definiuje sygnałów, więc używana jest raczej jako wybór.

signallist operator**=**moneta**,** wybór**,** zwrot**;**

W sieci Petriego należy zdefiniować kolor typu rekord[1)], który zawiera typy sygnałów znajdujących się na liście.

3. Podsumowanie

W artykule przedstawiono najważniejsze fragmenty modelowania konstrukcji obiektowych z zastosowaniem sieci Petriego. Główną przesłanką do tworzenia algorytmu tłumaczenia konstrukcji obiektowych UML była możliwość wykorzystania istniejących narzędzi sprawdzania właściwości sieci Petriego. Z tego powodu nałożone zostało ograniczenie, aby tworzenie nowych obiektów nie modyfikowało topologii sieci. Wydaje się, że przyjęte rozwiązanie, spełniając ten warunek, nie wprowadza ograniczeń modelowania diagramów UML, a przez to może być stosowane w szerokiej klasie modelowanych systemów.

Generowany w wyniku zaproponowanego algorytmu model wyrażony w języku sieci Petriego może być stosowany do analizy właściwości systemu definiowanego z wykorzystaniem języka UML. Należy go jednak traktować jako pewne ramy specyfikacji, w które wpisywane są inne modele (np. odpowiednie sieci opisujące diagramy stanów lub diagramy czynności). Wynikowy model może być wówczas weryfikowany względem właściwości wymaganych dla konstruowanej aplikacji. Dokładniejszy opis translacji oraz proponowanego postępowania zamieszczono w [6].

Literatura

[1] Rumbaugh J., Jacobsen I., Booch G.: *The Unified Modeling Language Reference Manual*. Second Edition, Addison-Welsley, tłum. Na jęz. polski: UML przewodnik użytkownika, WNT 2001
[2] Dokumentacja środowiska Telelogic Tau 2.4
[3] Standard UML 2.0, http://www.uml.org/#UML2.0 (http://www.omg.org/docs/formal/07-02-05.pdf)
[4] Jensen K.: *Coloured Petri nets: Basic concepts, analysis methods and practical use*. Springer 1996
[5] Dokumentacja środowiska CPN Design 4.0, http://www.daimi.au.dk/designCPN/man/
[5] Szmuc W.: *Modelowanie wybranych diagramów języka UML 2.0 z zastosowaniem kolorowanych sieci Petriego*. Kraków, AGH 2007 (rozprawa doktorska)

[1)] Użycie uni byłoby w tym przypadku rozwiązaniem efektywniejszym, jednak pojawiły się problemy z wyłuskaniem zmiennych – CPN zgłaszał niezgodność typów.