Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ

ROZPRAWA DOKTORSKA

MGR INŻ. WOJCIECH SZMUC

MODELOWANIE WYBRANYCH DIAGRAMÓW JĘZYKA UML 2.0 Z ZASTOSOWANIEM KOLOROWANYCH SIECI PETRIEGO

PROMOTOR:
dr hab. Marcin Szpyrka, prof. AGH

Kraków 2014

AGH

University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science
and Biomedical Engineering

DEPARTMENT OF APPLIED COMPUTER SCIENCE

PHD THESIS

MGR INŻ. WOJCIECH SZMUC

MODELLING OF SELECTED UML 2.0 DIAGRAMS WITH COLOURED PETRI NETS

SUPERVISOR:
Marcin Szpyrka, DSc, PhD

Krakow 2014

Pamięci Marii Antoniny i Piotra
Kazimierza Lewińskich — moich
dziadków

Spis treści

1. Wprowadzenie 7

1.1. Cel badań i teza pracy 9

1.2. Zawartość pracy 10

2. Wprowadzenie do sieci Petriego 11

2.1. Sieci miejsc i przejść (Place/Transition nets) 11

2.2. Sieci kolorowane (Coloured Petri Nets) 13

2.3. Czasowe sieci kolorowane (Timed CP-nets) 19

2.4. Hierarchiczne sieci kolorowane (Hierarchical CP-nets) 24

3. Wprowadzenie do UML 28

3.1. Algorytm konwersji UML $\rightarrow$ CPN 29

3.2. Predefiniowane typy danych 30

3.3. Ogólne konstrukcje języka 32

3.4. Wspólne symbole 35

3.5. Relacje w UML 35

3.6. Modelowanie pakietów 37

4. Algorytm translacji modelu ogólnej specyfikacji systemu 40

4.1. Modelowanie przypadków użycia 40

4.2. Modelowanie scenariuszy 40

5. Translacja ogólnego modelu zachowania systemu 64

5.1. Modelowanie czynności 64

5.2. Modelowanie zachowania 71

6. Algorytm translacji modelu szczegółowej specyfikacji systemu 90

6.1. Modelowanie klas 90

6.2. Modelowanie architektury 100

6.3. Modelowanie komponentów 102

6.4. Modelowanie rozmieszczenia 103

7. Zaawansowane konstrukcje UML 105

7.1. Rozszerzalność 105

7.2. Klasy metamodelu ... 106

8. Podsumowanie ... 109

8.1. Wnioski końcowe ... 110

8.2. Perspektywy dalszych badań ... 111

8.3. Podziękowania ... 111

A. Dodatek ... 112

1. Wprowadzenie

Wraz ze wzrostem obszarów zastosowań systemów informatycznych, wzrastają oczekiwania względem przedsiębiorstw wytwarzających oprogramowanie. Co raz częściej stawiane są pozornie sprzeczne żądania, aby systemy informatyczne były rozwijane szybko i jednocześnie charakteryzowały się wysoką niezawodnością działania. Biorąc pod uwagę wydajność współcześnie stosowanych rozwiązań sprzętowych, w wielu przypadkach większy nacisk stawiany jest na niezawodność systemu niż jego wydajność. Przykładem systemów informatycznych, dla których szczególny nacisk stawiany jest na eliminację błędów, są systemy krytyczne ze względu na bezpieczeństwo (ang. safety critical systems [34]). Terminem tym określane są systemy, których błędy działania mogą spowodować istotne straty ekonomiczne, uszkodzenia fizyczne, czy też mogą mieć wpływ na zdrowie lub życie ludzi oraz na środowisko naturalne [35], [38], [39], [43].

Istotny wpływ na końcową jakość systemu informatycznego ma przyjęta metodologia jego wytwarzania, na którą składają się zbiór czynności i związanych z nimi wyników, które prowadzą ostatecznie do finalnego produktu. Niezależnie od przyjętej metodologii koszt eliminacji błędów systemu informatycznego jest tym wyższy im później błędy te zostają wykryte [38]. Warto dodać, że koszt usuwania błędów wykrytych na etapie wdrożenia, może być nawet 500 razy większy niż koszt usuwania błędów wykrytych na etapie projektowania [38].

W literaturze opisanych wiele różnych podejść do wytwarzania oprogramowania [38], [41], [42], przy czym pewne etapy takie jak analiza, projektowanie, czy implementacja są wspólne dla nich wszystkich. Współcześnie etap analizy i projektowania oprogramowania został zdominowany przez język UML (Unified Modeling Language [7], [37], [11], [14]). Język UML powstał w 1997 roku jako efekt połączenia prac Jamesa Rumbaugha, Grady'ego Boocha oraz Ivara Jacobsona, którzy wcześniej indywidualnie rozwijali języki graficznego modelowania oprogramowania. Współcześnie język UML 2.0 jest najbardziej popularnym językiem modelowania oprogramowania i jest rozwijany przez konsorcjum OMG (Object Management Group). UML umożliwia przedstawienie systemu z różnych perspektyw (architektura, zachowanie) zależnych od modelowanego aspektu. Poszczególne perspektywy modelowane są z zastosowaniem przypisanych do nich diagramów. UML wspomaga tworzenie spójnego modelu składającego się z różnych diagramów. Nie wyklucza to nadmiarowości opisu systemu dopóki model jest spójny. Jest to szczególnie przydatna cecha, ponieważ diagramy mogą zawierać informacje z kolejnych etapów budowy oprogramowania. Pozwala to na ujęcie cyklu rozwijania oprogramowania, od modelu wymagań po testowanie kodu, w obrębie jednego środowiska. UML umożliwia bieżącą weryfikację niesprzeczności kolejnych uściśleń

zapewniając w ten sposób utrzymanie rozwoju systemu w kierunku zgodnym z wymaganiami użytkownika. W niniejszej rozprawie wykorzystano język UML 2.0 w wersji zaimplementowanej w narzędziu TAU 2.4 firmy Telelogic AB.

Język UML jest stosowany nie tylko do modelowania rozwijanego oprogramowania, ale również do modelowania procesów biznesowych, reprezentowania struktur organizacyjnych itp. Uniwersalność tworzonych rozwiązań, możliwość rozszerzania języka poprzez definiowanie profili itp. wykluczają w zasadzie możliwość zdefiniowania jednoznacznej semantyki tego języka modelowania. Brak ten wyklucza m.in. możliwość bezpośredniej formalnej analizy modeli opracowanych w języku UML. Warto dodać, że stosowanie metod formalnych bywa narzucane wymaganiami prawnymi, np. przy ocenie bezpieczeństwa systemów teleinformatycznych stosowane są normy ITSEC (*Information Technology Security Evaluation Criteria*) i *Common Criteria* (norma ISO 15408), które w zależności od poziomu bezpieczeństwa zalecają lub wymagają zastosowania metod formalnych. Między innymi te przesłanki spowodowały, że wielu naukowców podjęło próby opracowania metody translacji modeli zapisanych w języku UML do wybranego formalnego języka modelowania, który pozwala na automatyczną weryfikację modelu. Wśród docelowych języków formalnych można znaleźć m.in. algebry procesów [36] (mCRL2), [20] (CSP), automaty czasowe [16], [31], język Z [12], sieci Petriego [5], [4], [25], [40].

Zdecydowana większość autorów prac tego typu wybiera jedną z klas sieci Petriego [33], [6], [9], [30], [22], [23], [50], [52] jako formalizm docelowy, przy czym częściej są to klasy zaliczane do sieci wysokiego poziomu. W prezentowanym w rozprawie podejściu wybrano klasę kolorowanych sieci Petriego [22], [23], [50], z zachowaniem składni wspieranej przez środowisko CPN Tools [24]. Wybór kolorowanych sieci Petriego można uzasadniać m.in. następującymi cechami tego formalizmu:

– Reprezentacja graficzna sieci kolorowanych jest łatwa do zrozumienia nawet przez osoby nie znające szczegółów teorii sieci Petriego. Jednocześnie w wielu miejscach jest ona podobna do elementów występujących na niektórych typach diagramów języka UML.

– Kolorowane sieci Petriego w równym stopniu pozwalają opisywać zarówno stany systemu, jak i jego akcje. W zależności od potrzeb istnieje możliwość skupienia się na jednym bądź drugim z tych aspektów.

– Definicja kolorowanych sieci Petriego nie jest zbyt rozbudowana, a wiele z zawartych w niej elementów spotyka się w matematyce oraz popularnych językach programowania. Jednocześnie własności tych sieci mogą być analizowane z użyciem różnych metod, włączając w to techniki weryfikacji modelowej [3], [8], [13], [51].

– Dostępne mechanizmy hierarchizacji modelu pozwalają na niezależne rozwijanie różnych fragmentów systemu, łączonych następnie w całość końcowej fazie konstruowania modelu [22], [49], [50].

– Czasowe rozszerzenie sieci kolorowanych pozwala na używanie ich do modelowania systemów czasu rzeczywistego [22], [44], [47], [48].

1.1. Cel badań i teza pracy

Proponowane w niniejszej rozprawie podejście wspiera proces modelowania oprogramowania z użyciem języka UML, umożliwiając weryfikację poprawności budowanego systemu z użyciem kolorowanych sieci Petriego [22], [23], [46], [45], [50], [53]. Takie rozwiązanie powinno zapewnić znacznie wyższą jakość budowanych systemów ze względu na eliminowanie błędów na wczesnych etapach rozwijania aplikacji, obniżając koszty modyfikacji koniecznych w przypadku pojawienia się problemów w późniejszych etapach. Formalny model pozwala również na zweryfikowanie całościowej funkcjonalności co może być nierealne w przypadku tworzenia scenariuszy testowych.

Podsumowując, przyjęte tezy pracy można ściśle sformułować następująco: *Zastosowanie kolorowanych sieci Petriego umożliwia efektywne wspomaganie rozwijania poprawnego oprogramowania przez formalną analizę poprawności modelowanych artefaktów języka UML 2.0 we wczesnych (analiza i projektowanie) fazach procesu wytwarzania.*

Powyższe tezy zostaną wykazane poprzez:

– Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu sekwencji oraz ogólnego diagramu interakcji.

– Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu stanów oraz diagramu przepływu.

– Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu klas, diagramu architektury oraz diagramu stanów.

– Wykorzystanie opracowanych algorytmów do tłumaczenia przykładu.

Warto podkreślić, że proponowane podejście znacznie różni się od rozwiązań spotykanych w literaturze. Przedstawiane rozwiązania zazwyczaj skupiają się na pojedynczych wybranych typach diagramów języka UML, a opracowana translacja ma za zadanie zdefiniowanie formalnej semantyki tych diagramów. Przykładowo w pracy doktorskiej M. Szlenka [40] skupiono się na określeniu formalnej semantyki dla diagramu klas języka UML. J. Jacobs i A. Simpson w pracy [20] opisują translację diagramów sekwencji do algebry procesów CSP [18], [17]. E. Kerkouche i inni w pracy [25] opisują translację diagramów UML do kolorowanych sieci Petriego, przy czym skupiają się niemal wyłącznie na diagramach stanów. Inaczej ma się sytuacja w przypadku pracy I. Obera i innych, gdzie formalizmem docelowym są automaty czasowe [1], [2], [32] zaś podstawą do translacji są wybrane elementy diagramów klas i stanów. W przeciwieństwie do wspomnianych rozwiązań, proponowane podejście bardziej kompleksowo traktuje modele opracowane w języku UML. Podstawą generowanych modeli w postaci kolorowanych sieci Petriego jest aż 6 typów diagramów języka UML.

Warto jeszcze wspomnieć o podejściach zorientowanych na narzędzia do weryfikacji modelowej. W niektórych przypadkach wybór docelowego formalizmu podyktowany jest formatem wejściowym narzędzi użytych później do weryfikacji formalnej modelu. Zazwyczaj stosowane są tutaj

techniki weryfikacji modelowej [3], [8], przy czym własności modeli są wyrażane z użyciem jednej z wybranych logik temporalnych [13], [15], [27], [26]. Jako preferowane narzędzia weryfikacji modeli wybierane są m.in Kronos [55] (np. [10]), SPIN [19] (np. [28]) i UPPAAL [21] (np. [29]).

1.2. Zawartość pracy

Pomijając Wprowadzenie niniejsza rozprawa została podzielona na 7 kolejnych rozdziałów i jeden dodatek, w którym przedstawiono przykład zastosowania opisanego podejścia. Szczegółowy opis zawartości poszczególnych rozdziałów podano poniżej.

– Rozdział 2 zawiera wprowadzenie do sieci Petriego. Zostały w nim opisane zagadnienia związane z sieciami miejsc i przejść, sieciami kolorowanymi, czasowymi oraz hierarchicznymi. Przedstawiono również matematyczny zapis poszczególnych pojęć umożliwiający ich formalną analizę.

– Rozdział 3 jest wprowadzeniem do języka UML (*Unified Modeling Language*). Przedstawiono w nim ogólny opis języka, jego zastosowanie oraz krótką charakterystykę poszczególnych rodzajów diagramów wraz z przedstawieniem możliwości konwersji na sieci Petriego. Jest to również początek opisu algorytmu konwersji konstrukcji UML na kolorowane sieci Petriego.

– Rozdział 4 przedstawia algorytm translacji dla modelu ogólnej specyfikacji systemu. Jest to początkowa faza budowania systemu z wykorzystaniem diagramów UML. Opisane zostały konstrukcje związane z modelowaniem przypadków użycia oraz scenariuszy.

– Rozdział 5 przedstawia algorytm translacji ogólnego modelu zachowania systemu. Wykorzystywane w nim pojęcia umożliwiają projektowanie dynamicznej strony modelowanego systemu na różnych poziomach ogólności.

– Rozdział 6 opisuje algorytm translacji modelu szczegółowej specyfikacji systemu. Przedstawione w nim diagramy pozwalają na modelowanie statycznego widoku tworzonej aplikacji.

– Rozdział 7 opisuje zaawansowane konstrukcje UML. Zostały w nim przedstawione rozwiązania umożliwiające rozszerzenie funkcjonalności modelowania. Omówione zostały również propozycje oraz problemy translacji na sieci Petriego.

– Rozdział 8 jest podsumowaniem przedstawionych we wcześniejszych rozdziałach zagadnień przez zaprezenowanie wniosków oraz propozycji dalszych badań.

2. Wprowadzenie do sieci Petriego

Sieci Petriego są jednym z najczęściej stosowanych narzędzi używanych do opisu i formalnej analizy systemów współbieżnych. Wśród wielu aktualnie istniejących mutacji sieci Petriego, podstawowym i pierwszym historycznie modelem były sieci miejsc i przejść [33], [30], [50]. Są one zaliczane do tzw. sieci Petriego niskiego poziomu, gdyż stan (znakowanie) sieci wyrażany jest wyłącznie poprzez liczbę znaczników zgromadzonych w danym miejscu, bez rozróżniania znaczników między sobą. Przedstawicielem sieci wysokiego poziomu są kolorowane sieci Petriego [22], [23], [50]. Notacja graficzna tych sieci jest taka sama jak dla sieci niskiego poziomu, ale została ona połączona z językiem programowania wysokiego poziomu, w rozważanym w rozprawie przypadku jest to CPN ML, oparty o funkcyjny język programowania Standard ML [54]. W sieciach tych znaczniki mają swój tych i wartość. Język CPN ML służy do definiowania typów znaczników używanych w sieci oraz inskrypcji elementów sieci, które są niezbędne do opisu przepływu tak zdefiniowanych znaczników. W niniejszym rozdziale przedstawiono podstawowe informacje na temat sieci miejsc i przejść oraz kolorowanych sieci Petriego.

2.1. Sieci miejsc i przejść (Place/Transition nets)

Sieci miejsc i przejść (PT-sieci) są najpopularniejszą klasą sieci Petriego niskiego poziomu. Ich popularność wynika między innymi z prostej definicji i dostępności dużej liczby łatwych do stosowania metod formalnej analizy własności sieci [50].

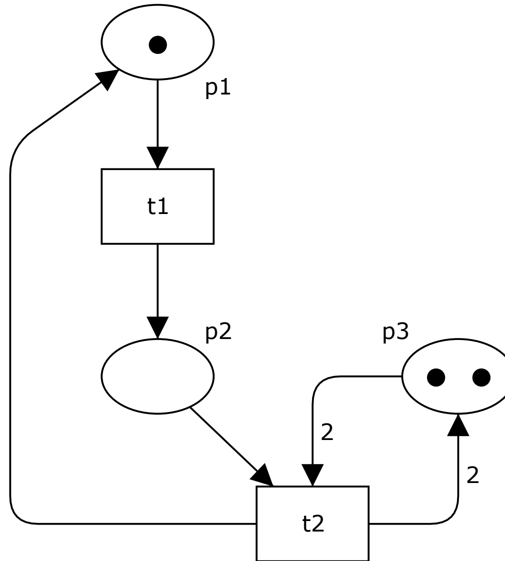
Definicja 2.1. *Siecią miejsc i przejść* (PT-siecią) nazywamy piątkę $PM = (P, T, A, W, s_0)$, gdzie:

1) P jest skończonym zbiorem miejsc (ang. *places*);
2) T jest skończonym zbiorem przejść (ang. *transitions*);
3) $A \subseteq P \times T \cup T \times P$ jest zbiorem łuków;
4) $W\colon A \to \mathbb{N}$ jest funkcją wag przypisująca etykiety (liczby naturalne) do każdego łuku;
5) $s_0\colon P \to \mathbb{N}^*$ jest funkcją opisująca oznakowanie początkowe (ang. *initial marking*), gdzie $\mathbb{N}^*$ oznacza zbiór liczb całkowitych nieujemnych.

Zakłada się ponadto, że dla każdej PT-sieci spełnione są warunki: $P \cap T = \emptyset$ i $P \cup T \neq \emptyset$.

Sieć miejsc i przejść jest graficznie przedstawiana jako graf dwudzielny, którego zbiór węzłów zawiera podzbiory miejsc i przejść. Łuki łączą węzły różnych typów, zaś waga łuku zastępuje moż-

liwość wielokrotnego łączenia łukami tych samych dwóch węzłów. Działanie sieci polega na przepływie znaczników między miejscami sieci. Znaczniki są zabierane z miejsc wejściowych w liczbie określonej przez wagę łuku wejściowego i wstawiane do miejsc wyjściowych przejścia w liczbie określonej przez wagę łuku wyjściowego.



Rysunek 2.1: Przykład sieci Petriego

Warunkiem koniecznym realizacji przejścia (zmiany stanu sieci) jest obecność odpowiedniej liczby znaczników w miejscach sieci. Liczbę znaczników w każdym miejscu określa funkcja stanu (oznakowanie). Szczególną rolę pełni tu funkcja oznakowania początkowego. Przykładową sieć miejsc i przejść przedstawiono na rysunku 2.1. W miejscu $p1$ znajduje się jeden znacznik, a w miejscu $p3$ dwa. Łuki połączone z miejscem $p3$ mają wagę 2, wszystkie pozostałe łuki sieci mają domyślną wagę równą 1. Na rysunku 2.1 przedstawiono tę samą sieć, lecz po wykonaniu przejścia $t1$. Przejście $t2$ nie zostało wyzwolone, ponieważ nie było znacznika w miejscu $p2$.

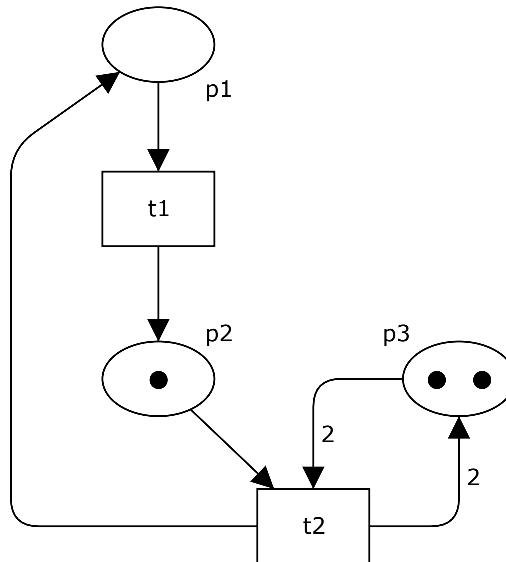
Zmiana stanu sieci

W celu uproszczenia zapisu określone zostanie pewne domknięcie funkcji W oznaczane $\overline{W}$. Dla dowolnej sieci $PM = (P, T, A, W, s_0)$ funkcja $\overline{W}\colon P \times T \cup T \times P \to \mathbb{N}^*$ jest przedłużeniem W określonym następująco

$$\overline{W} = \begin{cases} W(x)\colon x \in A \\ 0\colon x \notin A \end{cases} \tag{2.1}$$

Definicja 2.2. Niech $PM = (P, T, A, W, s_0)$ będzie siecią Petriego, określoną jak w definicji 2.1.

1. Zbiór wszystkich oznakowań tej sieci oznaczamy symbolem S. Zawiera on wszystkie funkcje postaci $s\colon P \to \mathbb{N}^*$.



Rysunek 2.2: Stan sieci po wykonaniu przejścia $t1$

2. Funkcja częściowa $\delta\colon S \times T \to S$ jest określona następująco:

$$Dom(\delta) = \{(s,t)\colon \forall p \in P\ s(p) \geqslant \overline{W}(p,t)\}. \tag{2.2}$$

Jeśli $(s,t) \in Dom(\delta)$, to:

$$\forall p \in P\ s'(p) = \delta(s,t)(p) = s(p) - \overline{W}(p,t) + \overline{W}(t,p). \tag{2.3}$$

Dla dowolnych s, s', t, jeśli $s' = \delta(s,t)$, to fakt ten będziemy zapisywać jako $s[t\rangle s'$ lub $s \xrightarrow{t} s'$.

Punkt pierwszy powyższej definicji określa zbiór wszystkich możliwych oznakowań sieci Petriego. Oznakowanie (stan) sieci określa ile znaczników znajduje się w poszczególnych miejscach sieci. Przepływ znaczników powoduje zmiany stanu sieci, wyznaczone przez funkcję częściową δ[1]. Warunkiem koniecznym wykonania przejścia t przy oznakowaniu s jest istnienie w każdym miejscu wejściowym wystarczającej liczby znaczników ($s(p) \geqslant \overline{W}(p,t)$). Oznakowanie musi zatem zapewniać odpowiednią liczbę znaczników w miejscach wejściowych dla danego przejścia, tak aby możliwe było pobranie tych znaczników w liczbie określonej przez funkcję etykietującą łuki W. Zmianę stanu wskutek wykonania przejścia opisuje drugi punkt definicji. Znaczniki są pobierane z miejsc wejściowych w liczbie określonej przez wartość funkcji W dla odpowiedniego łuku (od miejsca do przejścia) oraz dodawane do miejsc wyjściowych w liczbie wyznaczonej przez tę samą funkcję dla odpowiednich łuków wyjściowych.

2.2. Sieci kolorowane (Coloured Petri Nets)

Formalna definicja kolorowanej sieci Petriego wymaga wprowadzenia pojęcia wielozbioru. W porównaniu do klasycznych zbiorów, dowolny element w wielozbiorze może wystąpić więcej

[1] Funkcja częściowa jest określona na pewnym podzbiorze to znaczy, jeśli $f\colon X \to Y$, to dziedzina tej funkcji $Dom(f) \subseteq X$.

niż raz. Definiując wielozbiór określa się jakie elementy zawiera i w ilu egzemplarzach. Wielozbiór można traktować jako zbiór z rozszerzoną funkcją przynależności.

Definicja 2.3. *Wielozbiorem* (*multi-set*) m nad niepustym zbiorem X nazywamy dowolną funkcję $m\colon X \to \mathbb{N}^*$. Dowolna nieujemna liczba całkowita $m(x) \in \mathbb{N}^*$ jest liczbą wystąpień elementu x w tym wielozbiorze. Wielozbiór jest zazwyczaj reprezentowany w postaci pewnej sumy:

$$\sum_{x\in X} m(x)`x. \tag{2.4}$$

Przyjmiemy ponadto następujące oznaczenia:

- X_{MS} oznacza zbiór wszystkich wielozbiorów nad zbiorem X;
- liczbę $m(x)$ nazywamy *współczynnikiem* elementu x;
- element $x \in X$ należy do wielozbioru m wtw, gdy $m(x) \neq 0$, fakt ten zapisujemy $x \in m$.

Interpretując powyższą definicję można stwierdzić, że wielozbiór składa się z dwóch części: zbioru elementów X oraz funkcji m określającej liczbę wystąpień elementów ze zbioru X. Dodanie lub usunięcie elementu z wielozbioru powoduje zmianę odpowiedniego współczynnika, a więc zmianę funkcji m. Wielozbiór zapisuje się często w postaci sumy elementów zbioru X poprzedzonych współczynnikami, np.: $2`a + 3`b$.

Dodawanie, mnożenie skalarne, porównywanie i wielkość wielozobiorów są definiowane następująco:

Definicja 2.4. Niech $m, m_1, m_2 \in X_{MS}$ i niech $n \in \mathbb{N}^*$. *Sumą wielozbiorów* m_1 i m_2 nazywamy wielozbiór:

$$m_1 + m_2 = \sum_{x\in X} (m_1(x) + m_2(x))`x. \tag{2.5}$$

Iloczynem wielozbioru m_1 *przez stałą* n nazywamy wielozbiór:

$$n \cdot m = \sum_{x\in X} (n \cdot m(x))`x. \tag{2.6}$$

Relację *równości* wielozbiorów definiujemy następująco:

$$m_1 = m_2 \Leftrightarrow \forall x \in X\colon m_1(x) = m_2(x). \tag{2.7}$$

Relację *mniejszości* wielozbiorów definiujemy następująco:

$$m_1 \leqslant m_2 \Leftrightarrow \forall x \in X\colon m_1(x) \leqslant m_2(x). \tag{2.8}$$

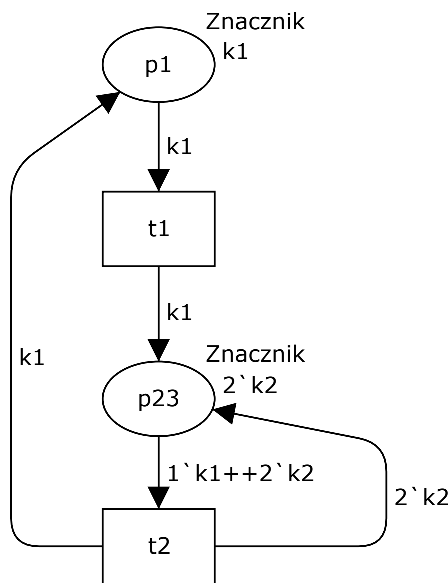
Jeżeli $m_2 \leqslant m_1$, to *różnicą wielozbiorów* m_1 i m_2 nazywamy wielozbiór:

$$m_1 - m_2 = \sum_{x\in X} (m_1(x) - m_2(x))`x. \tag{2.9}$$

Rozmiarem wielozbioru m nazywamy liczbę:

$$|m| = \sum_{x\in X} m(x). \tag{2.10}$$

Pojęcie kolorowanej sieci Petriego wprowadzono w celu zmniejszenia stopnia skomplikowania sieci miejsc i przejść. Podstawowa korzyść polega na połączeniu fragmentów o podobnej budowie. W celu rozróżnienia poszczególnych fragmentów „koloruje się” znaczniki umożliwiając w ten sposób sformułowanie warunków (wyrażenia na łukach, dozorów) gwarantujących odpowiedni przepływ znaczników. Ponieważ może wystąpić wiele znaczników danego koloru do opisu sieci kolorowanej używa się wielozbiorów. Na rysunku 2.3 przedstawiono przykład kolorowanej sieci Petriego otrzymanej przez przekształcenie sieci z rysunku 2.1. W tym przypadku połączono miejsca $p2$ i $p3$. Funkcjonalną tożsamość obu schematów uzyskuje się przez pokolorowanie znaczników i ustawienie odpowiednich wag na łukach. Procedurę przekształcenia opisano krótko poniżej.



Rysunek 2.3: Przykład sieci kolorowanej

Na początku został zdefiniowany kolor mogący przyjmować jedną z dwu wartości $k1$ albo $k2$:

```
colset Znacznik = with k1 | k2;
```

Następnie ustawiono odpowiednie wagi na łukach definiując liczbę i typ znaczników przepływających przez łuk (operator ++ jest stosowany do opisu wielozbiorów w programie CPN Tools, którego użyto do zaprojektowania omawianej sieci). Określono liczbę oraz typ znaczników w stanie początkowym (etykieta z opisem wielozbioru umieszczona obok miejsca). Ustalono również kolor miejsca będący typem (kolorem) znaczników, które mogą się w nim znajdować (etykieta z nazwą koloru umieszczona obok miejsca). Ponieważ na rysunku 2.3 nie pokazano wszystkich możliwych konstrukcji, niektóre z nich zostaną przedstawione poniżej.

Dozór jest wyrażeniem, które zezwala na wyzwolenie przejścia tylko przez znaczniki określonego typu. Jest to warunek umieszczony w pobliżu przejścia zawierający się w kwadratowych nawiasach. Przykładowo dozór dla przejścia $t1$ mógłby mieć postać `[k=k1]`, przy wcześniejszym zadeklarowaniu zmiennej k typu *znacznik* (`var k: Znacznik;`) i przypisaniu jej do łuku wejściowego przejścia (aby konstrukcja miała sens).

Wyrażenie przypisane do łuku ustala typ i parametry znaczników zwracanych przez przejście.

W tym celu może używać słów kluczowych języka CPN ML [54]. Najczęściej stosowana jest konstrukcja: **if** <warunek> **then** <wyrażenie 1> **else** <wyrażenie 2>, gdzie: *warunek* jest wyrażeniem logicznym, *wyrażenie 1* jest wyrażeniem wykonywanym, gdy warunek jest prawdziwy, a *wyrażenie 2* jest wyrażeniem wykonywanym, gdy warunek nie jest prawdziwy. Niekiedy dla określonego warunku nie powinien być oddawany znacznik. W takiej sytuacji stosuje się słowo kluczowe **empty**.

Warto podkreślić, że chociaż w pracy do opisu elementów sieci kolorowanych będzie używany język CPN ML, to nie stanowi on elementu formalnej definicji sieci. Sama definicja jest niezależna od języka, który stosujemy do definiowania typów danych, zmiennych, czy też definiowania wyrażeń łuków lub dozorów dla przejść. Wybór CPN ML wynika z faktu, że jest on stosowany w oprogramowaniu CPN Tools, które jest używane do projektowania sieci opisywanych w rozprawie. Inne narzędzia komputerowe mogą używać inny język do definiowania inskrypcji sieci.

Formalna definicja sieci kolorowanej zostanie poprzedzona wprowadzeniem pojęć pomocniczych. Po pierwsze zakładamy, że dostępny jest *zbiór typów danych* Σ i zbiór V *zmiennych* typów ze zbioru Σ. Dla dowolnej zmiennej $v \in V$, *typ zmiennej* oznaczamy symbolem $\mathit{Type}[v]$.

Niech Expr oznacza zbiór wyrażeń dostarczany przez język wybrany do definiowania inskrypcji sieci. Dla dowolnego wyrażenia $e \in \mathit{Expr}$, *typ wyrażenia*, tj. typ wartości uzyskanej w wyniku ewaluacji wyrażenia, oznaczamy symbolem $\mathit{Type}[e]$. Zbiór wolnych zmiennych występujących w wyrażeniu $e \in \mathit{Expr}$ oznaczamy symbolem $\mathit{Var}[e]$. Dla zbioru zmiennych $V' \subseteq V$, zbiór wyrażeń, takich że $\mathit{Var}[e] \subseteq V'$ oznaczamy symbolem $\mathit{Expr}_{V'}$.

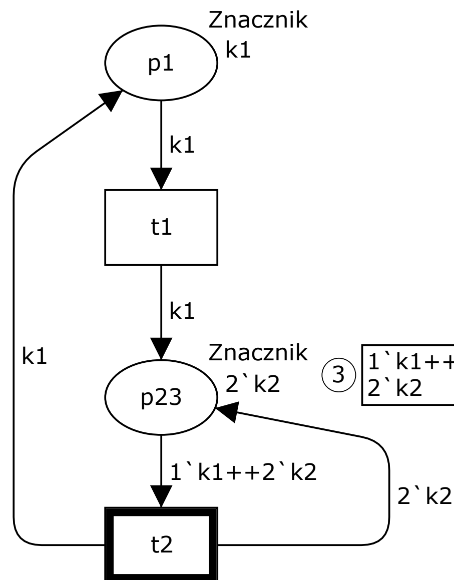
Definicja 2.5. *Niehierarchiczną kolorowaną siecią Petriego* (CP-siecią) nazywamy krotkę $\mathit{CPN} = (P, T, A, \Sigma, V, C, G, E, I)$, gdzie:

1) P jest skończonym zbiorem *miejsc*;
2) T jest skończonym zbiorem *przejść* takim, że $P \cap T = \emptyset$;
3) $A \subseteq (P \times T) \cup (T \times P)$ jest skończonym zbiorem *łuków*;
4) Σ jest skończonym zbiorem *typów* (*kolorów*), będących zbiorami niepustymi;
5) V jest skończonym zbiorem *zmiennych* takich, że $\mathit{Type}[v] \in \Sigma$ dla wszystkich zmiennych $v \in V$.
6) $C\colon P \to \Sigma$ jest funkcją kolorów (ang. *color set function*), która przypisuje typ znaczników do każdego z miejsc;
7) $G\colon T \to \mathit{Expr}_V$ jest funkcją dozorów (ang. *guard function*), która każdemu z przejść $t \in T$ przypisuje *dozór* taki, że $\mathit{Type}[G(t)] = \mathit{Bool}$.
8) $E\colon A \to \mathit{Expr}_V$ jest *funkcją wyrażeń łuków* (ang. *arc expression function*), która każdemu łukowi $a \in A$ przypisuje wyrażenia takie, że $\mathit{Type}[E(a)] = C(p)_{MS}$, gdzie p jest miejscem z którym połączony jest łuk a.
9) $I\colon P \to \mathit{Expr}_\emptyset$ jest *funkcją inicjalizacji* (ang. *initialisation function*), która każdemu z miejsc $p \in P$ przypisuje *wyrażenie inicjalizujące* takie, że $\mathit{Type}[I(p)] = C(p)_{MS}$.

Załączona definicja została zaczerpnięta z monografii [23]. Różni się ona nieznacznie od definicji zawartych np. w monografiach [22], [50] – nie są dopuszczone łuki wielokrotne i jawnie

dołączono do krotki zbiór zmiennych. Podejście takie jest jednak bardziej czytelne i odpowiada podejściu jakie stosowane jest przy projektowaniu CP-sieci w oprogramowaniu CPN Tools.

Niektóre elementy powyższej definicji zostały już przedstawione przy omawianiu rysunku 2.3. Pozostałe zostaną tu pokrótce omówione. Funkcja dozorów G jest stosowana, aby przez przejście mogły przechodzić tylko określone znaczniki. Musi ona operować na zmiennych należących do zbioru kolorów i zwracać wartość typu logicznego ($Bool$). Jeżeli dozór nie został zdefiniowany, to przyjmuje się jego wartość domyślną tj. stałą $true$. Warunki określone dla funkcji E oznaczają, że wartościowanie wyrażenia łuku musi być wielozbiorem nad kolorem, który przypisano do miejsca, z którym łuk jest połączony. W przypadku funkcji inicjalizującej, wymagane jest, aby dla każdego miejsca oznakowanie początkowe było wielozbiorem nad kolorem przypisanym do tego miejsca.



Rysunek 2.4: Oznakowanie sieci z rysunku 2.3 po wykonaniu przejścia $t1$

Poniżej przeanalizowano zagadnienia związane z wykonywaniem kolorowanej sieci Petriego. Rysunek 2.4 przedstawia oznakowanie sieci po wykonaniu przejścia $t1$. W przypadku miejsc zawierających znaczniki, ich aktualna liczba wyświetlana jest w postaci liczby umieszczonej w okręgu, zaś aktualne oznakowanie w postaci dodatkowej etykiety. W rozważanym stanie, w miejscu $p1$ nie ma znaczników, natomiast miejsce $p2$ zawiera 3 znaczniki. Pogrubienie tranzycji $t2$ wskazuje na możliwość jej wykonania w tym stanie.

Formalny opis przebiegu zmiany stanu sieci kolorowanej wymaga wprowadzenia dodatkowych pojęć i oznaczeń.

Definicja 2.6. Dla CP-sieci $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ definiujemy następujące pojęcia:

1. *Oznakowaniem* (ang. *marking*) nazywamy dowolną funkcję M, która każdemu z przejść $p \in P$ przypisuje wielozbiór znaczników $M(p) \in C(p)_{MS}$.
2. *Znakowanie początkowe* M_0 jest znakowaniem uzyskanym w wyniku wartościowania wyrażeń inicjalizujących, $M_0(p) = I(p)\langle\rangle$ dla dowolnego $p \in P$.

3. *Zbiorem zmiennych przejścia* t nazywamy zbiór wolnych zmiennych występujących w dozorze przejścia t oraz w wyrażeniach wszystkich łuków wejściowych i wyjściowych tego przejścia. Zbiór ten oznaczamy symbolem $Var(t) \subseteq V$.
4. *Wiązaniem* (ang. *binding*) przejścia t nazywamy funkcję b, która każdej zmiennej $v \in Var(t)$ przypisuje wartość $b(v) \in Type[v]$. Zbiór wszystkich wiązań przejścia t oznaczamy symbolem $B(t)$.
5. *Elementem wiążącym* (ang. *binding element*) nazywamy dowolną parę (t, b), gdzie $t \in T$ oraz $b \in B(t)$. Zbiór wszystkich elementów wiążących przejścia t jest zdefiniowany jako $BE(t) = \{(t, b) \colon b \in B(t)\}$. Zbiór wszystkich elementów wiążących w modelu CPN oznaczamy symbolem BE.
6. *Krokiem* (ang. *step*) $Y \in BE_{MS}$ nazywamy dowolny niepusty wielozbiór elementów wiążących.

Pojęcia aktywności i wykonania są ściśle powiązane z ewaluacją dozorów i wyrażeń łuków. Każda taka ewaluacja jest rozpatrywana zawsze w kontekście pewnego wiązania (wartościowania zmiennych). Dla elementu wiążącego (t, b) symbolem $G(t)\langle b\rangle$ oznaczamy wynik ewaluacji dozoru $G(t)$ przy wiązaniu b. Podobnie symbolem $E(a)\langle b\rangle$ oznaczamy wynik ewaluacji wyrażenia łuku a przy wiązaniu b.

Dla dowolnego miejsca p symbolem $E(p, t)$ oznaczamy wyrażenie łuku prowadzącego od miejsca p do przejścia t. Jeżeli taki łuk nie istnieje, to przyjmujemy, że $E(p, t) = \emptyset_{MS}$. Podobnie definiujemy wyrażenie $E(t, p)$ łuku prowadzącego od przejścia t do miejsca p.

Definicja 2.7. Element wiążący $(t, b) \in BE$ jest *aktywny* (ang. *enable*) przy oznakowaniu M wtw., gdy spełnione są warunki:

1) $G(t)\langle b\rangle = true$,
2) $\forall p \in P \colon E(p, t)\langle b\rangle \leqslant M(p)$.

Jeżeli element wiążący (t, b) jest aktywny przy oznakowaniu M, to może zostać *wykonany* prowadząc do znakowania M' zdefiniowanego jako:

$$\forall p \in P \colon M'(p) = M(p) - E(p, t)\langle b\rangle + E(t, p)\langle b\rangle. \tag{2.11}$$

Definicja 2.8. Krok $Y \in BE_{MS}$ jest *aktywny* przy oznakowaniu M wtw, gdy spełnione są warunki:

1) $\forall (t, b) \in Y \colon G(t)\langle b\rangle = true$,
2) $\forall p \in P \colon \sum_{(t,b)\in Y} E(p, t)\langle b\rangle \leqslant M(p)$.

Jeżeli krok Y jest aktywny przy oznakowaniu M, to może zostać *wykonany* prowadząc do znakowania M' zdefiniowanego jako:

$$\forall p \in P \colon M'(p) = M(p) - \sum_{(t,b)\in Y} E(p, t)\langle b\rangle + \sum_{(t,b)\in Y} E(t, p)\langle b\rangle. \tag{2.12}$$

Jeżeli wykonanie kroku Y (elementu wiążącego (t, b)) powoduje zmianę znakowania z M na M', to mówimy wówczas, że oznakowanie M' jest *bezpośrednio osiągalne* z M, co zapisujemy jako $M \xrightarrow{Y} M'$ ($M \xrightarrow{(t,b)} M'$).

Niech krok Y będzie aktywny przy oznakowaniu M. Jeśli $(t_1, b_1), (t_2, b_2) \in Y$ oraz $(t_1, b_1) \neq (t_2, b_2)$, to (t_1, b_1) oraz (t_2, b_2) są *równolegle aktywne* (ang. *concurrently enabled*). Przejście t jest współbieżnie aktywne z samym sobą jeśli $|Y(t)| \geqslant 2$, gdzie $Y(t)$ oznacza wielozbiór wiązań przejścia t należących do kroku Y. To samo dotyczy elementu wiążącego (t, b), który jest współbieżnie aktywny z samym sobą jeśli $Y(t, b) \geqslant 2$.

Warto zauważyć, że wykonanie przejścia nie jest określone wyłącznie przez jego wybór, lecz zależy również od wiązania. W zależności od wyboru wiązania (b), dane przejście może zostać wykonane na kilka sposobów przy zadanym oznakowaniu M. Właściwość ta wynika z ukrycia rozgałęzień sieci miejsc i przejść w zmiennych, funkcjach przypisujących i kolorach. „Rozwinięcie" sieci kolorowanej w odpowiadającą jej sieć miejsc i przejść, spowoduje odtworzenie struktury i wówczas dane przejście rozdzieli się na kilka innych, które mogą być współbieżnie aktywne przy pewnych oznakowaniach.

Definicja 2.9. *Skończonym ciągiem wykonań* o długości $n \geqslant 0$ nazywamy ciąg oznakowań i kroków

$$M_1 \xrightarrow{Y_1} M_2 \xrightarrow{Y_2} M_3 \ldots M_n \xrightarrow{Y_n} M_{n+1} \tag{2.13}$$

taki, że $M_i \xrightarrow{Y_i} M_{i+1}$ dla dowolnego $1 \geqslant i \geqslant n$.

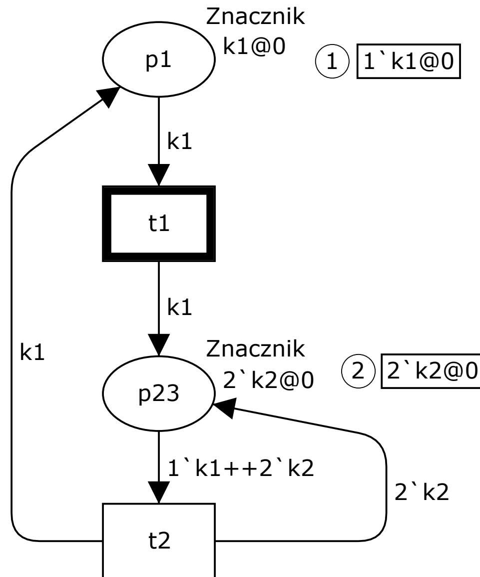
O każdym z oznakowań występujących w ciągu (2.13) mówimy, że jest ono *osiągalne* z M_1. Analogicznie można zdefiniować nieskończony ciągiem wykonań. Zbiór oznakowań osiągalnych z oznakowania M oznaczamy symbolem $\mathcal{R}(M)$. W szczególności interesuje nas zbiór $\mathcal{R}(M_0)$.

2.3. Czasowe sieci kolorowane (Timed CP-nets)

Istnieje wiele sposobów wprowadzenia czasu do sieci Petriego [50]. Ze względu na stosowane narzędzie (CPN Tools) zostanie przedstawiony tylko jeden z nich. Polega on na przypisywaniu znacznikom pieczątek czasowych. Określają one do jakiego momentu czasu znacznik musi pozostać w konkretnym miejscu, aby był gotowy do użycia, tzn. pobrany do wykonania przejścia. Znacznik z pieczątką może zostać pobrany z miejsca, w którym się znajduje, tylko pod warunkiem, że jego pieczątka czasowa ma wartość mniejszą lub równą stanowi globalnego zegara. Algorytm zmiany stanu sieci czasowej realizowany jest następująco: jeśli w aktualnym momencie czasowym (stanie zegara) pewne wiązania są przygotowane (przejścia możliwe do wykonania), to są one realizowane, bez zmiany stanu zegara. Po wyczerpaniu wszystkich takich wiązań, stan zegara jest zwiększany do momentu odpowiadającego spełnieniu najwcześniejszego wiązania.

W związku z pojawieniem się dodatkowych atrybutów konwencja zapisu kolorowanej sieci Petriego zawiera pewne rozszerzenia umożliwiające modelowanie czasu. Jednym z nich są wspominane już pieczątki czasowe. Oznacza się je przy pomocy symbolu po którym określona jest wartość pieczątki. Przykładowo wyrażenie $k1@3$ oznacza, że znacznik $k1$ może być pobrany z miejsca, w

którym się znajduje, gdy stan zegara osiągnie wartość 3 lub większą. Odmienna konstrukcja stosowana jest przy nadawaniu pieczątek czasowych. W takim przypadku po symbolu umieszcza się znak $+$ po którym następuje wartość o jaką należy zwiększyć pieczątkę czasową. Przykład sieci kolorowanej z czasem znajduje się na rysunku 2.5.



Rysunek 2.5: Przykład sieci kolorowanej z czasem (czas: 0)

Przypisanie pieczątek czasowych do znaczników jest formalnie zapisywane jako odpowiednie rozszerzenie elementów wielozbiorów. Niech $\mathbb{T}$ oznacza zbiór dopuszczalnych wartości globalnego zegara ($\mathbb{T} = \mathbb{N}^*$).

Definicja 2.10. *Wielozbiorem czasowym* (ang. *timed multiset*) tm nad niepustym zbiorem X nazywamy funkcję $tm\colon X \times \mathbb{T} \to \mathbb{N}^*$, taką że suma

$$tm(x) = \sum_{t \in \mathbb{T}} tm(x,t) \tag{2.14}$$

jest skończona dla każdego $x \in X$. Wartość $tm(x)$ oznacza liczbę wystąpień elementu x w wielozbiorze tm.

Listą pieczątek czasowych (ang. *timestamp list*) elementu x nazywamy listę

$$tm[x] = [t_1, t_2, \ldots, t_{tm(x)}] \tag{2.15}$$

spełniającą warunek $t_i \leqslant t_{i+1}$ dla dowolnych $1 \leqslant i < tm(x)$. Lista ta zawiera wartości czasowe t, dla których $tm(x,t) > 0$ i każda wartość t występuje na liście $tm(x,t)$ razy.

Wielozbiory czasowe będziemy zapisywać w postaci sumy

$$\sum_{x \in X} tm(x)`x@tm[x] \tag{2.16}$$

Zbiór wszystkich wielozbiorów nad zbiorem X oznaczamy X_{TMS}. Dowolna nieujemna liczba całkowita $tm(x)$, gdzie $x \in X$ nazywana jest współczynnikiem czasowego wielozbioru tm. Element $x \in X$ należy do czasowego wielozbioru, jeśli $t \neq 0$. Fakt ten zapisujemy jako $x \in tm$. Każdy czasowy wielozbiór $tm \in X_{TMS}$ wyznacza wielozbiór $tm_U \in X_{MS}$, określony następująco:

$$tm_U = \sum_{x \in X} tm(x)`x \tag{2.17}$$

Jak wspomniano wielozbiory są najczęściej reprezentowane w postaci sumy. Przykładowo wielozbiór czasowy $1`k1@[3] + 2`k2@[0,0]$ opisuje trzy znaczniki, gdzie pierwszy jest gotowy do pobrania począwszy od momentu czasu 3, natomiast dwa pozostałe od razu (moment czasowy 0). W przypadku programu CPN Tools rozważany wielozbiór czasowy jest zapisywany jako `` 1`k1@3 +++ 2`k2@0 ``.

Warunki wzbudzenia są określone przez relację $\leqslant$ między pewnym wartościowaniem wyrażenia łuku a odpowiednim wielozbiorem umieszczonym w miejscu wejściowym. W praktyce jest to więc porównywanie odpowiednich wielozbiorów. W przypadku sieci bez czasu, zagadnienie to nie nastręcza trudności. Dla sieci czasowych oprócz porównywania ilości odpowiednich znaczników musimy jeszcze sprawdzić zachowanie relacji dla pieczątek czasowych. Zagadnienie to jest złożone, gdyż te same znaczniki mogą mieć przypisany różny czas, co więcej, porównywane listy mogą być różnej długości.

Relację $\leqslant$ oraz operację odejmowania dla wielozbiorów czasowych wprowadza się rozszerzając odpowiednią relację i operację (oznaczane tak samo jak dla wielozbiorów nieczasowych). Należy zwrócić uwagę, że pozostawienie tylko części nieczasowej, przy żądaniu równości pieczątek czasowych odpowiadających tym samym kolorom jest niewystarczające, gdyż przy usuwaniu znaczników zazwyczaj mamy do czynienia z różnymi czasami i interesujące jest usunięcie elementu który ma czas mniejszy lub równy względem określonego. To ostatnie wymaganie oznacza bowiem, że znacznik jest gotowy do pobrania. W celu zdefiniowania operacji porównania i odejmowania dla wielozbiorów czasowych najpierw zostaną one określone dla list reprezentujących pieczątki czasowe. Nie będą wprowadzane odrębne symbole dla definiowanych pojęć – ich dziedzina będzie wynikać z kontekstu.

Dla czasowych wielozbiorów nad zbiorem X i zbioru wartości czasowych $\mathbb{T}$ operacje porównania, odejmowania i dodania czasu dla list pieczątek czasowych definiujemy następująco.

Definicja 2.11. Niech $tm[x] = [t_1, t_2, \ldots, t_{tm(x)}]$, $tm_1[x] = [t_1^1, t_2^1, \ldots, t_{tm_1(x)}^1]$, $tm_2[x] = [t_1^2, t_2^2, \ldots, t_{tm_2(x)}^2]$ będą listami pieczątek czasowych pewnego elementu $x \in X$.

1. $tm_1[x] \leqslant_{\mathbb{T}} tm_2[x]$ wtw, gdy $tm_1(x) \leqslant tm_2(x)$ i $t_i^1 \geqslant t_i^2$ dla wszystkich $1 \leqslant i \leqslant tm_1(x)$.
2. Dla $t \in \mathbb{T}$ takiego, że $t \geqslant t_1$, $tm[x] -_{\mathbb{T}} t$ jest listą pieczątek czasowych:

   – $[t_1, t_2, t_3, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{tm(x)}]$,

   gdzie i jest największym indeksem dla którego $t_i \leqslant t$.
3. Jeżeli $tm_1[x] \leqslant_{\mathbb{T}} tm_2[x]$, to $tm_1[x] -_{\mathbb{T}} tm_2[x]$ jest listą pieczątek czasowych:

– $tm_1[x] -_{\mathbb{T}} tm_2[x] = ((([t_1^2, t_2^2, \ldots, t_{tm_2(x)}^2] -_{\mathbb{T}} t_1^1) -_{\mathbb{T}} t_2^1) \cdots -_{\mathbb{T}} t_{tm_1(x)}^1)$.

4. Dla $t \in \mathbb{T}$, $tm[x]_{+t}$ jest listą pieczątek czasowych:

   – $tm[x]_{+t} = [t_1 + t, t_2 + t, \ldots, t_{tm(x)} + t]$.

Dla czasowych wielozbiorów nad zbiorem X operacje dodawania, porównania, odejmowania i dodania czasu definiujemy następująco.

Definicja 2.12. Niech $tm, tm_1, tm_2 \in X_{TMS}$.

1. $\forall (x, t) \in X \times \mathbb{T}\colon (tm_1 + tm_2)(x, t) = tm_1(x, t) + tm_2(x, t)$.
2. $tm_1 \leqslant tm_2$ wtw, gdy $\forall x \in X\colon tm_1[x] \leqslant_{\mathbb{T}} tm_2[x]$.
3. Jeżeli $tm_1 \leqslant tm_2$, to $tm_1 - tm_2$ jest wielozbiorem czasowym spełniającym warunki:
   – $\forall x \in X\colon (tm_1 - tm_2)(x) = tm_1(x) - tm_2(x)$;
   – $\forall x \in X\colon (tm_1 - tm_2)[x] = tm_1[x] -_{\mathbb{T}} tm_2[x]$;
4. Dla $t \in \mathbb{T}$ wielozbiór czasowy tm_{+t} jest zdefiniowany jako:
   – $\forall x \in X\colon tm_{+t}(x) = tm(x)$ i $tm_{+t}[x] = tm[x]_{+t}$.

Składnia czasowych CP-sieci jest zbliżona do składni sieci nieczasowych. Różnica dotyczy wyłącznie kwestii czasowych. Typy znaczników mogą być definiowane jako *czasowe* lub *nieczasowe*. Miejsca, którym przypisano czasowy typ znaczników nazywamy *miejscami czasowymi*, a miejsca, którym przypisano nieczasowy typ znaczników, *miejscami nieczasowymi*. Łuki sieci dzielimy również na *czasowe* i *nieczasowe*, zależnie od tego, czy są połączone z miejscami czasowymi, czy z miejscami nieczasowymi.

Definicja 2.13. *Niehierarchiczną czasową kolorowaną siecią Petriego* nazywamy krotkę $CPN_T = (P, T, A, \Sigma, V, C, G, E, I)$, gdzie:

1) P jest skończonym zbiorem *miejsc*.
2) T jest skończonym zbiorem *przejść* takim, że $P \cap T = \emptyset$.
3) $A \subseteq (P \times T) \cup (T \times P)$ jest skończonym zbiorem *łuków*.
4) Σ jest skończonym zbiorem *typów* (*kolorów*), będących zbiorami niepustymi; każdy z typów jest zdefiniowany jako *czasowy* lub *nieczasowy*.
5) V jest skończonym zbiorem *zmiennych* takich, że $Type[v] \in \Sigma$ dla wszystkich zmiennych $v \in V$.
6) $C\colon P \to \Sigma$ jest funkcją kolorów, która przypisuje typ znaczników do każdego z miejsc. Miejsce p nazywamy *miejscem czasowym*, jeżeli $C(p)$ jest typem czasowym, w przeciwnym przypadku miejsce p nazywamy *nieczasowym*.
7) $G\colon T \to Expr_V$ jest funkcją dozorów, która każdemu z przejść $t \in T$ przypisuje *dozór* taki, że $Type[G(t)] = Bool$.
8) $E\colon A \to Expr_V$ jest *funkcją wyrażeń łuków*, która każdemu łukowi $a \in A$ przypisuje wyrażenia takie, że:

– $\mathit{Type}[E(a)] = C(p)_{MS}$, jeżeli miejsce p, z którym połączony jest łuk a, jest miejscem nieczasowym;

– $\mathit{Type}[E(a)] = C(p)_{TMS}$, jeżeli miejsce p, z którym połączony jest łuk a, jest miejscem czasowym.

9) $I\colon P \to \mathit{Expr}_{\emptyset}$ jest *funkcją inicjalizacji*, która każdemu z miejsc $p \in P$ przypisuje *wyrażenie inicjalizujące* takie, że:

– $\mathit{Type}[I(p)] = C(p)_{MS}$, jeżeli miejsce p jest miejscem nieczasowym;

– $\mathit{Type}[I(p)] = C(p)_{TMS}$, jeżeli miejsce p jest miejscem czasowym.

W przypadku stosowania oprogramowania CPN Tools, pominięcie pieczątek czasowych w wyrażeniu inicjalizującym miejsca czasowego oznacza przyjęcie domyślnych wartości równych 0. Ponadto możliwe jest przypisanie inskrypcji czasowej do przejścia, co oznacza w praktyce dodanie jej do wszystkich wyrażeń czasowych łuków wyjściowych.

Pojęcia takie jak wiązanie, element wiążący i krok są definiowane dla sieci czasowych tak samo jak dla sieci nieczasowych. Dla sieci czasowych definiujemy dodatkowe pojęcia.

Definicja 2.14. Niech dana będzie czasowa CP-sieć $CPN_T = (P, T, A, \Sigma, V, C, G, E, I)$.

1. *Oznakowaniem* nazywamy dowolną funkcję M, która każdemu z przejść $p \in P$ przypisuje wielozbiór znaczników $M(p)$ taki, że:

   – $M(p) \in C(p)_{MS}$, jeżeli miejsce p jest miejscem nieczasowym;

   – $M(p) \in C(p)_{TMS}$, jeżeli miejsce p jest miejscem czasowym.

2. *Oznakowaniem czasowym* (ang. *time marking*) nazywamy parę (M, t^*), gdzie M jest oznakowaniem, a $t^* \in \mathbb{T}$ jest wartością zegara globalnego.
3. *Czasowe znakowanie początkowe* (ang. *initiam timed marking*) jest parą $(M_0, 0)$, gdzie $M_0(p) = I(p)\langle\rangle$ dla dowolnego $p \in P$.

Należy zwrócić uwagę na wprowadzenie pojęcia czasowego oznakowania, określonego przez oznakowanie (wielozbiory czasowe przypisane do miejsc) oraz aktualną wartość czasu (zegara). Jest to konieczne, gdyż samo oznakowanie nie jest wystarczające (dla sieci czasowej) do określenia jej zachowania. Wartość czasu może bowiem decydować o wzbudzeniu i sposobie wykonania przejścia.

Definicja 2.15. Krok $Y \in BE_{MS}$ jest *aktywny* w czasie t' przy oznakowaniu czasowym (M, t^*) wtw, gdy spełnione są warunki:

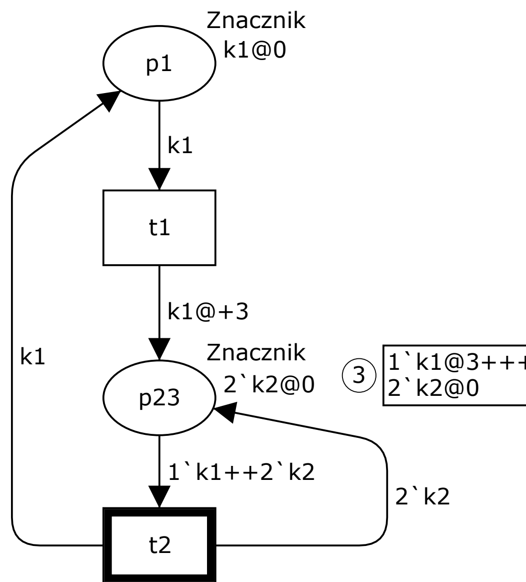
1) $\forall (t,b) \in Y : G(t)\langle b\rangle = true$,
2) $\sum_{(t,b)\in Y} E(p,t)\langle b\rangle \leqslant M(p)$, dla wszystkich nieczasowych miejsc p;
3) $\sum_{(t,b)\in Y} (E(p,t)\langle b\rangle)_{+t'} \leqslant M(p)$, dla wszystkich czasowych miejsc p;
4) $t^* \leqslant t'$;
5) t' jest najmniejszą wartością dla której istnieje krok spełniający warunki 1–4.

Jeżeli krok Y jest aktywny przy oznakowaniu czasowym (M, t^*) w czasie t', to może zostać *wykonany* w czasie t' prowadząc do znakowania czasowego (M', t') zdefiniowanego jako:

– $M'(p) = (M(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle) + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$, dla wszystkich nieczasowych miejsc p;

– $M'(p) = (M(p) - \sum_{(t,b)\in Y} (E(p,t)\langle b\rangle)_{+t'}) + \sum_{(t,b)\in Y} (E(t,p)\langle b\rangle)_{+t'}$, dla wszystkich czasowych miejsc p.

Pojęcia takie jak osiągalność, osiągalne oznakowania, sekwencje wystąpień są definiowane analogicznie jak dla sieci nieczasowych tyle tylko, że odnoszą się do oznakowań czasowych.

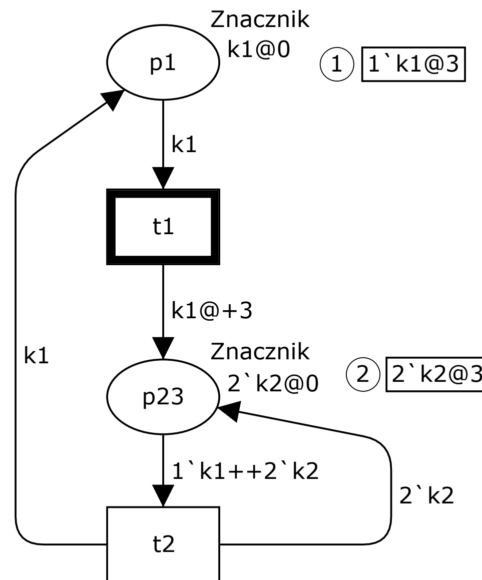


Rysunek 2.6: Sieć z rys. 2.5 po wykonaniu przejścia $t1$ (czas: 0)

Na rysunku 2.6 pokazano sieć po wykonaniu przejścia $t1$. Ponieważ do jego wykonania potrzebny był znacznik $k1$, którego pieczątka czasowa ma wartość 0, to może on zostać pobrany przy stanie zegara równym 0. Do pieczątki czasowej znacznika zostaje dodana wartość 3 (na łuku wyjściowym z przejścia). Powoduje to sytuację, w której, przy obecnym stanie zegara, żadne przejście nie może zostać zrealizowane. W takim przypadku stan zegara zostaje zwiększony do wartości (najmniejszej), przy której jakiekolwiek przejście będzie wyzwolone. Ponieważ sieć jest dość prosta można od razu zauważyć, że jest to wartość 3. Sytuację w której zegar został przestawiony na wartość 3 i zostało już wykonane przejście $t2$ przedstawia rysunek 2.7. Warto zauważyć, że możliwe jest jeszcze wykonanie przejścia $t1$.

2.4. Hierarchiczne sieci kolorowane (Hierarchical CP-nets)

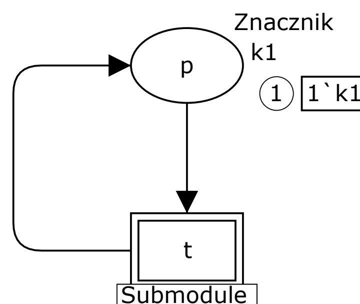
Sieci hierarchiczne stosuje się w celu ułatwienia zarządzania dużymi sieciami kolorowanymi. Dzięki właściwościom hierarchicznego opisu można efektywnie budować duże sieci składające się z modułów (podsieci).



Rysunek 2.7: Sieć z rys. 2.6 po wykonaniu przejścia $t2$ (czas: 3)

Podstawowym problemem przy budowaniu hierarchii jest zdefiniowanie połączeń między różnymi poziomami. Ze względu na charakter sieci Petriego, w ich wersji hierarchicznej, takim połączeniem mogą być miejsca albo przejścia. Biorąc po uwagę rodzaj połączeń, wyróżnia się dwa typy hierarchizacji sieci: przez *podstawienie przejść* (ang. *substitution of transitions*) oraz przez łączenie miejsc (ang. *fusion sets*). Zostaną one przedstawione w dalszym części podrozdziału.

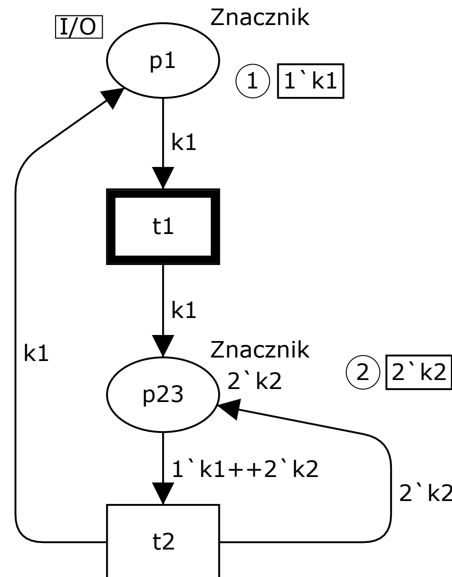
Użycie hierarchii umożliwia definiowanie modułów (nazywanych również stronami) tj. podział sieci na części i zapisanie jej fragmentów w poszczególnych modułach. Sieć hierarchiczna składa się z co najmniej dwóch modułów. Jeżeli dwa moduły są łączone z zastosowaniem konstrukcji podstawiania przejść, to znajdują się one na różnych poziomach abstrakcji i jeden z nich jest *podmodułem* drugiego. Każdy z modułów jest w pewnym sensie niehierarchiczną kolorowaną siecią Petriego, ale zawierającą dodatkowe elementy opisu, umożliwiające scalenie modułów w kompletną sieć.



Rysunek 2.8: Główny moduł sieci hierarchicznej

Główną przesłanką w metodzie podstawienia przejść jest ukrycie części sieci pod symbolem przejścia. Pozwala to schować szczegóły zagadnienia pozostawiając czytelną strukturę. Przykład modułu z podstawianym przejściem pokazano na rys. 2.8. W przypadku modelowania z użyciem CPN Tools, podstawiane przejścia są rysowane podwójną linią. Każde z takich przejść ma ponadto

przypisaną etykietę z nazwą podmodułu, który został z nim skojarzony. Należy podkreślić, że podstawiane przejścia nie mają przypisanych dozorów, a połączone z nimi łuki nie mają przypisanych wyrażeń. Szczegóły funkcjonalności ukrytej pod przedstawianym przejściem zawarte są w w skojarzonym z nim podmodule.



Rysunek 2.9: Podmoduł sieci hierarchicznej

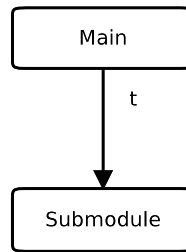
Podstawiane przejście można traktować jak wywołanie procedury, a skojarzony z nim moduł jako implementację tej procedury. Moduł skojarzony z podstawianym przejściem t pokazano na rys. 2.9. Jak widać jest to sieć identyczna jak przedstawiona na rys. 2.3 z wyjątkiem dodatkowej etykiety (I/O), skojarzonej z miejscem $p1$.

Miejsca wejściowe podstawianego przejścia nazywane są *gniazdami wejściowymi* (ang. *input sockets*), a miejsca wyjściowe *gniazdami wyjściowymi* (ang. *output sockets*). Jeżeli miejsce jest jednocześnie miejsce wejściowym i wyjściowym podstawianego przejścia, to mówimy o *gniazdach wejściowo-wyjściowych* (ang. *input/output sockets*). Gniazda podstawianego przejścia stanowią jego interfejs. W rozważanym przykładzie podstawiane przejście t ma jedno gniazdo wejściowo-wyjściowe p.

Interfejs podmodułu stanowią jego *porty* (ang. *ports*). Są to wyróżnione miejsca (oznacza się je dodatkowymi etykietami jak w przypadku miejsca $p1$ na rys. 2.9), które służą do wymiany znaczników z otoczeniem. *Porty wejściowe* służą do importu znaczników, a *porty wyjściowe* do eksportu. Możliwe jest definiowanie portów wejściowo-wyjściowych. Niezależnie od portów, moduł może zawierać swoje wewnętrzne miejsca, nie będące portami.

Połączenie modułu z podmodułem jest realizowane przez relację, której elementami są pary (gniazdo, port). Relacja ta wiąże interfejs modułu z interfejsem podmodułu. Porty wejściowe muszą był łączone z gniazdami wejściowymi, porty wyjściowe z gniazdami wyjściowymi itd. Dwa miejsca, które tworzą parę (gniazdo, port) stanowią w rzeczywistości jedno *miejsce złożone* (ang. *compound place*). Miejsca tworzące miejsce złożone muszą mieć przypisane takie same typy, a ich

wyrażenia inicjalizujące muszą dawać w wyniku ewaluacji takie same wielozbiory. W przypadku portów można pominąć wyrażenie inicjalizujące. W takim przypadku przejmują one znakowanie od skojarzonego z nimi gniazda.



Rysunek 2.10: Hierarchia modeluj

Struktura połączeń między modułami w sieci hierarchicznej przedstawiana jest za pomocą grafu skierowanego nazywane *hierarchią modelu* (ang. *model hierarchy*). Graf taki dla rozważanego modelu pokazano na rys. 2.10. Nazwy modułów są umieszczane wewnątrz węzłów, a łuki są etykietowane nazwami podstawianych przejść. Węzły bez łuków wejściowych reprezentują *moduły główne* (ang. *prime modules*). Hierarchia modułów musi być grafem acyklicznym.

W rozważanym przykładzie występują tylko dwa poziomy abstrakcji, ale w przypadku ogólnym może być ich dowolna skończona liczba. Możliwe jest także użycie tego samego modułu jako podmodułu dla kilku podstawianych przejść. W takim przypadku model zawiera kilka *instancji* takiego modułu, a każda tych instancji ma własne niezależne od innych znakowanie.

Drugim mechanizmem łączącym moduły są *fuzje miejsce* (ang. *fusion set*). Umożliwiają one na tworzenie miejsc złożonych, które są scaleniem dwóch lub większej liczby miejsca należących do różnych modułów lub instancji modułów. Wszystkie miejsca należące do takiej fuzji współdzielą jedno znakowanie. Muszą one mieć przypisany ten sam typ, a ich wyrażenia inicjalizujące muszą dawać w wyniku ewaluacji takie same wielozbiory. Miejsca wchodzące w skład fuzji są wyróżnione dodatkową etykietą z nazwą fuzji. Fuzje miejsc sa podobne do zmiennych globalnych występujących w wielu językach programowania.

Formalna definicja sieci hierarchicznych [23] wymaga zdefiniowania modułu, który jest siecią niehierarchiczną z wyróżnionym zbiorem podstawianych przejść i portów wraz z ich typami. Następnie definiowana jest sieć hierarchiczna jako sieć złożona ze zbioru modułów wraz z funkcjami, które określają przypisanie modułów do podstawianych przejść i przypisanie portów do gniazd oraz z wyróżnionym zbiorem fuzji.

Dynamikę sieci hierarchicznych definiuje się podobnie jak dla sieci niehierarchicznych, przy czym np. w miejsce zbioru miejsc rozważa się klasy abstrakcji relacji określonej w zbiorze zawierającym wszystkie instancje miejsc. Jeżeli dwie instancje miejsc znajdą się w tej samej samej klasie abstrakcji, to oznacza to, że albo należą do jednej fuzji miejsce, albo tworzą parę (gniazdo, port). W rzeczywistości matematyczny zapis definicji jest nieco bardziej zawiły, ale ich intuicyjne znaczenie pozostaje bez zmian. Pełny formalny opis sieci hierarchicznych można znaleźć np. monografii [23].

3. Wprowadzenie do UML

Język UML 2.0 wprowadza dodatkowe (w odniesieniu do wcześniejszych wersji) aspekty modelowania systemów czasu rzeczywistego przez możliwość użycia do opisu systemu odmiennej reprezentacji maszyny stanowej.

Opisując wykorzystanie UML 2.0 bazowano głównie na jego implementacji w narzędziu TAU 2.4 firmy Telelogic AB (aktualnie wewnątrz konsorcjum IBM). Wersja ta nie jest całkowicie zgodna ze standardem, lecz ponieważ jest dostarczana w postaci narzędzia wspomagającego modelowanie, wydaje się bardziej użyteczna niż wspomniany standard. Warto podkreślić, że system TAU 2.4 został opracowany jako dedykowany do modelowania systemów czasu rzeczywistego z zastosowaniem języka UML, co dodatkowo uzasadnia dokonany wybór. Należy ponadto wspomnieć o różnicach między poszczególnymi wersjami TAU, które wraz z rozwojem umożliwiają wykorzystanie większego zakresu konstrukcji. Na przykład do badań związanych z niniejszą pracą wykorzystywano początkowo wersję TAU 2.1, która w ograniczonym zakresie umożliwiała modelowanie przy użyciu diagramów sekwencji. W wersji 2.4 diagramy te umożliwiały już tworzenie bardziej skomplikowanych struktur (na przykład definiowanie zachowania ramki wplecionej). W nowszej wersji występują również diagramy, których nie ma w starszych wersjach (np. diagram aktywności). W pracy zajęto się głównie graficzną reprezentacją konstrukcji UML, ponieważ równoważna semantycznie reprezentacja tekstowa jest mniej czytelna.

UML umożliwia przedstawienie systemu z różnych perspektyw (architektura, zachowanie) zależnych od modelowanego aspektu. Poszczególne perspektywy modelowane są z zastosowaniem przypisanych do nich diagramów. UML wspomaga stworzenie spójnego modelu składającego się z różnych diagramów. Nie wyklucza to nadmiarowości opisu systemu dopóki model jest spójny. Jest to szczególnie przydatna cecha, ponieważ diagramy mogą zawierać informacje z kolejnych etapów budowy oprogramowania. Pozwala to na ujęcie cyklu rozwijania oprogramowania, od modelu wymagań po testowanie kodu, w obrębie jednego środowiska. UML umożliwia bieżącą weryfikację niesprzeczności kolejnych uścisleń zapewniając w ten sposób utrzymanie rozwoju systemu w kierunku zgodnym z wymaganiami użytkownika.

W celu zapewnienia czytelności pracy poszczególne elementy UML zostaną przedstawione wraz z ich translacją na sieć Petriego. Pozwoli to opisać odpowiadające sobie konstrukcje przedstawiając je „obok siebie”, co ułatwi zrozumienie zasad translacji. Dla ukazania możliwie szerokiego spektrum tłumaczonych konstrukcji przedstawionych zostanie kilka fragmentów z różnorodnych systemów. Przykłady wykorzystane dla wyjaśnienia translacji poszczególnych elementów nie zostaną

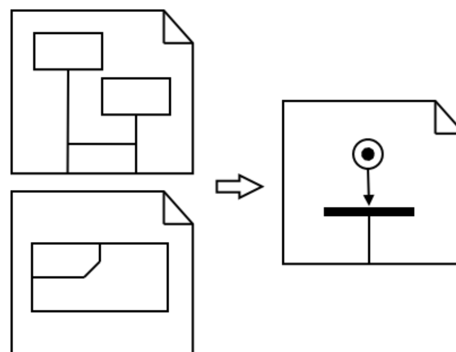
opisane w całości ze względu na ograniczony zakres wykorzystania ich funkcjonalności. Wydaje się jednak, że sposób opisu oraz powszechna znajomość opisywanych systemów powinna zapewnić zrozumiałość przedstawionych przykładów bez konieczności ich dokładnego opisywania.

3.1. Algorytm konwersji UML → CPN

Ze względu na czytelność zasad tłumaczenia diagramów UML na sieci Petriego przedstawione zostaną metody dotyczące poszczególnych konstrukcji (nie jest to całkowity algorytm). W związku z powyższym przy budowie sieci Petriego należy zwrócić uwagę na jej wymagania. Ponieważ sieć Petriego jest grafem dwudzielnym miejsca oraz przejścia muszą odpowiednio się przeplatać. Jeżeli konwersja prowadzi do złamania tej zasady to należy wstawić odpowiedni element (miejsce albo przejście) doprowadzając do zgodności z definicją:

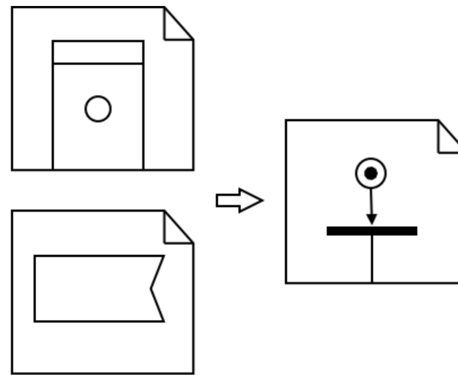
– Jeżeli brakuje miejsca to jest ono wstawiane. Jego typ (kolor) jest określony przez typ znacznika, który ma być przez nie przesyłany. Miejsce to nie ma znakowania początkowego.
– Jeżeli brakuje przejścia, to jest ono wstawiane (bez dozoru).
– Łuki wejściowe oraz wyjściowe mają etykiety związane z przesyłaniem odpowiedniego znacznika (wynikającego z istniejącego już fragmentu sieci).

Ze względu na różny (zależny od etapu projektowania) poziom ogólności diagramów UML wynikowa sieć Petriego ma odmienną konstrukcję. W rezultacie powstają niezależne modele reprezentowane przez sieci Petriego odzwierciedlające poszczególne stadia rozwijania systemu. Umożliwia to skonstruowanie oraz sprawdzenie formalnego modelu w aktualnie rozwijanym fragmencie. Jeżeli dla każdego etapu projektowania systemu tworzony jest model formalny w etapie końcowym otrzymuje się 3 reprezentacje opisujące różne poziomy złożoności systemu (patrz rysunek 3.1, 3.2 oraz 3.3).

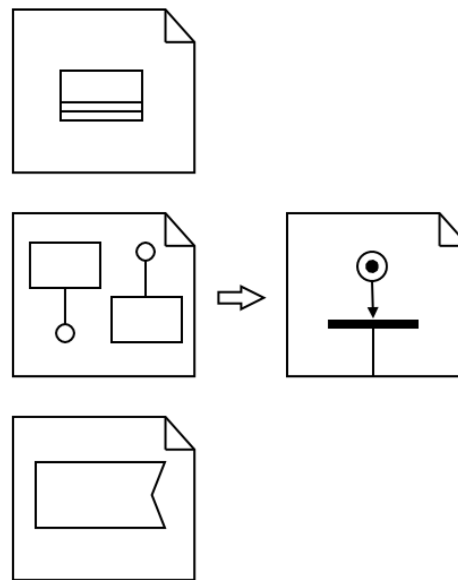


Rysunek 3.1: Generowanie sieci Petriego na podstawie diagramu sekwencji oraz ogólnego diagramu interakcj

Należy zauważyć, że niektóre rodzaje diagramów mogą być używane wymiennie do definiowania fragmentów systemu. Muszą one jednak umożliwiać opis z odpowiednim poziomem szczegółowości.



Rysunek 3.2: Generowanie sieci Petriego na podstawie diagramu stanów oraz diagramu przepływu



Rysunek 3.3: Generowanie sieci Petriego na podstawie diagramu klas, diagramu architektury oraz diagramu stanów

3.2. Predefiniowane typy danych

Konstrukcje modelowania danych w UML umożliwiają definiowanie danych w różnorodny sposób. UML nie zawiera wielu wbudowanych typów danych, oferuje jednak możliwość rozszerzenia o różne zestawy typów danych zależnie od dziedziny aplikacji. Uzyskuje się to przez definiowanie typów danych w bibliotekach modelu (często opisywanych jako predefiniowane pakiety). Występują 3 rodzaje zestawów predefiniowanych danych, które mogą zostać użyte:

Predefined jest pakietem zawierający podstawowe typy danych wraz z możliwymi operacjami. Pakiet *Predefined* jest właściwie rozszerzeniem UML, które jest zawsze dostępne w projekcie. Pakiet definiuje wiele typów danych, oraz kilka innych dodatków. Pakiet *Predefined* określa między innymi typy danych występujące w OMG UML (jak *Integer*, *Boolean* itp.), lecz również operacje na tych typach, co już nie jest określone w standardzie. Dla każdego typu danych występuje zestaw operacji, które mogą być wykorzystane w wyrażeniach z tym typem.

Poniżej znajduje się zestawienie definicji opisywanego pakietu wraz z ich odpowiednikami w sieci Petriego:

- *Boolean* (typ logiczny) – w deklaracji koloru ze słowem kluczowym `bool`.
- *Character* (typ znakowy) – w deklaracji koloru łańcuch jednoznakowy:
  `colset C = string with " " .. "~" and 1 .. 1;`
- *String* (typ łańcuchowy) – jest rodzajem kolekcji implementowanej jako uporządkowana lista (albo łańcuch) elementów określonego typu:
  `colset S = list typZmiennej;`
  gdzie `typZmiennej` jest zdefiniowanym wcześniej kolorem. Konstrukcja listy w sieci Petriego nie zapewnia pełnej odpowiedniości z typem *String*. Może się, więc okazać, że należy uzupełnić funkcjonalność przy pomocy dodatkowych fragmentów sieci. Innym rozwiązaniem jest wykorzystanie konstrukcji zaproponowanych do modelowania tablicy. Typ *String* jest domyślnie wykorzystywany jako kontener instancji, których liczebność jest większa niż 1. Szczególnym przypadkiem są instancje znaków, które mogą w sieci Petriego zostać zapisane w zmiennej typu (koloru) *string*:
  `colset S = string;`
  Należy więc rozróżniać sposób wykorzystania zmiennej łańcuchowej i od niego uzależniać rodzaj konwersji. Łańcuch w UML przypomina tablicę w C, która również reprezentuje łańcuch.
- *Charstring* (łańcuch znaków) – synonim *String*.
- *Integer* (liczby całkowite) – w deklaracji koloru słowo kluczowe `int`.
- *Natural* (liczby naturalne) – w deklaracji koloru zawężenie zakresu całkowitego:
  `colset I = int with 0 .. 1073741823;`
- *Real* (liczby zmiennoprzecinkowe) – w deklaracji koloru słowo kluczowe `real`[1]
- *Array* (typ tablicowy) – w sieci Petriego nie występuje typ tablicowy. Ogólnie konstrukcję tablicy można zamodelować przy pomocy znaczników, przy czym każdy znacznik jest pojedynczym wierszem (krotką o rozmiarze równym liczbie kolumn). Taka konstrukcja powoduje rozproszenie tablicy, co należy uwzględnić podczas analizy. Alternatywą w niektórych przypadkach może być wykorzystanie listy jako konstrukcji zastępującej tablicę. Jeżeli tablica ma mało rekordów można ją zamodelować przy pomocy krotki w ten sposób, że każdy element krotki jest pojedynczym rekordem.
- *Any* (dowolny typ) – typ opisany w punkcie 5.2 — Modelowanie zachowania.
- *const* (stała – modyfikator na początku definicji typu) – deklaracja bez nazwy typu np.:
  `val s = 1;`

Aby zmienna w sieci Petriego była dostępna, w danym miejscu musi znajdować się w znaczniku. Z powyższego wynika, że znacznik powinien być rekordem, którego pola zawierają wszystkie

[1]Zastosowanie oprogramowania CPN Tools wskazuje, że typ ten nie został zaimplementowany, mimo że występuje w dokumentacji.

wymagane w danym fragmencie sieci zmienne. Deklaracja koloru znacznika może wyglądać następująco:

```
colset znacznik = record pole1 : typ1 * pole2 : typ2 *...
```

przy czym `typ1`, `typ2`, ... są wcześniej zdefiniowanymi kolorami.

- *PLUS_INFINITY* (dodatnia nieskończoność) – jest stałą rzeczywistego typu danych, określającą największą wartość rzeczywistą, która może być użyta w określonym systemie.
- *MINUS_INFINITY* (ujemna nieskończoność) – jest stałą rzeczywistego typu danych określającą najmniejszą wartość rzeczywistej, która może być użyta w określonym systemie.

Profil TTDRTTypes jest pakietem zawierającym typy danych oraz operacje obsługiwane tylko przez weryfikator modelu (Model Verifier) oraz generatora kodu C.

Pakiet zwiera również predefiniowany sygnał *none*. Jest to wbudowany w system sygnał używany do modelowania niedeterministycznego zachowania. Jest on używany tylko w abstrakcyjnych specyfikacjach lub modelu przeznaczonym do symulacji (testowania) a nie dla modeli, z których mają być zbudowane aplikacje. Sygnał *none* wykorzystywany jest jako wyzwalacz w definicji niedeterministycznego (spontanicznego) przejścia. Przyczyna modelowanego zdarzenia nie może być kontrolowana ani pod względem momentu w którym się pojawia, ani częstotliwości.

W sieci Petriego *none* może być zamodelowany przez konstrukcję składającą się z 2 przejść i jednego miejsca. Miejsce w opisywanej konstrukcji jest wejściem oraz wyjściem dla obu przejść. Jedno z przejść wysyła znacznik jako sygnał odpowiadający modelowanej funkcjonalności. Miejsce jest zainicjowane jednym znacznikiem. Wybór odpalanego przejścia jest niedeterministyczny zgodnie z definicją sieci Petriego.

Profil TTDCppPredefined [2] – jest pakietem zawierającym typy danych oraz operacje obsługiwane jedynie przez generator aplikacji C++.

3.3. Ogólne konstrukcje języka

Metodologia obiektowa wykorzystuje specyficzne pojęcia (np. widoczność), których tłumaczenie na sieci Petriego może doprowadzić do jej nadmiernego skomplikowania (sieć Petriego nie wspiera podejścia obiektowego[3]). Wydaje się jednak, że mechanizmy sprawdzania poprawności UML umożliwiają skontrolowanie tych właściwości w dostateczny sposób (standardowe mechanizmy kompilatorów). W związku z ograniczeniami sieci Petriego definicje tekstowe umieszcza się w globalnym węźle deklaracji (odpowiednik symbolu tekstowego). Należy więc zwrócić uwagę żeby nie powtarzały się wykorzystywane nazwy. Można to osiągnąć przez dodanie na końcu nazwy UML numer strony sieci odpowiadającej definicji związanej z daną nazwą.

[2] W związku z brakiem opisu pakietu nie stworzono algorytmu konwersji do sieci Petriego.

[3] Istnieją wprawdzie definicje obiektowych sieci Petriego, jednak formalizm ten zatrzymał się w początkowej fazie rozwoju [http://www.llpn.com/OPNs.html].

Należy zwrócić uwagę, że UML (tak, jak Java w przeciwieństwie do C++), jako metodologia w pełni obiektowa, nie umożliwia tworzenia zmiennych globalnych nie będących obiektami.

Odwołanie przez kwalifikację nazwy (`nazwanaPrzestrzeń::element`) jest w sieci Petriego realizowane analogicznie do powiązania opisanego w punkcie 3.5 — Relacje w UML.

Pojęciami specyficznymi dla metodologii obiektowej są również te związane z wirtualnością. Odnosi się ona do elementów, które są zawarte w typach (klasyfikatory, które mogą być specjalizowane). Gdy kontener jest specjalizowany, indywidualna wirtualność każdego zawieranego elementu określa czy może on zostać zmieniony. Jeżeli zawierany element jest wirtualny (*virtual*) jest możliwe przedefiniowanie (zmiana) tego elementu, gdy jego kontener jest specjalizowany.

Jeżeli element w specjalizowanym kontenerze został przedefiniowany (*redefined*) zmienia się definicja oryginalnego elementu z kontenera bazowego. Oryginalny element w bazowym kontenerze musi być wirtualny. Przedefiniowany element jest nadal wirtualny to oznacza, że jeżeli kontener jest specjalizowany jeszcze raz, element może znowu zostać przedefiniowany.

Jeżeli element w specjalizowanym kontenerze jest sfinalizowany (*finalized*) zmienia to definicję oryginalnego elementu z bazowego kontenera. Oryginalny element w kontenerze bazowym musi być wirtualny. Sfinalizowanie wiąże się również z zabronieniem dalszych przedefiniowań tego elementu, jeżeli kontener jest specjalizowany jeszcze raz. W tym sensie sfinalizowany znaczy „przedefiniowany ale nie wirtualny".

Jeżeli zawierany element nie jest wirtualny nie jest możliwe jego przedefiniowanie (zmiana) gdy kontener jest specjalizowany.

Tłumaczenie rezultatów użycia powyższych modyfikatorów na sieci Petriego jest związane z relacją uogólnienia przedstawioną w punkcie 3.5 — Relacje w UML. Opisane rozwiązanie wynika z faktu, że w UML nie występuje typ wskaźnikowy, więc nie ma potrzeby weryfikacji typu elementu wskazywanego przez wskaźnik (w przeciwieństwie do C++).

Mimo, że sprawdzenie poprawności zastosowania powyższych konstrukcji również pozostaje w UML, jednak wnoszą one pewne modyfikacje do architektury, które muszą znaleźć odzwierciedlenie w sieci Petriego. Algorytm konwersji takich elementów na sieć Petriego wymaga skopiowania całego fragmentu sieci przedstawionego na danym diagramie i jego odpowiednich modyfikacji. Polegają one na zmianie (w kopii) sieci bazowej na sieć odpowiadającą specjalizowanemu kontenerowi. W ten sposób otrzymuje się statyczny ekwiwalent specjalizowanego kontenera.

Element może mieć też inne cechy:

– *wyprowadzany* (*derived*) – wyliczany na podstawie innych elementów. Wiąże się to z istnieniem algorytmu, który pozwala na obliczenie elementu. W sieci Petriego odpowiada to konstrukcji w której algorytm jest zawarty na stronie podstawionej pod przejście. Jest ono używane (wysyłany jest do niego znacznik), gdy potrzebna jest wartość elementu wyprowadzanego. Wynik jest zwracany przez znacznik wychodzący z przejścia. Taka konstrukcja zapewnia jednakowość algorytmu dla wszystkich wywołań.

– *zewnętrzny* (*external*) – oznacza, że element znajduje się poza modelem – nie jest dla niego generowany kod. Ponieważ w modelu nie ma definicji elementu, sieć Petriego nie jest dla niego generowana.

– *teoretyczny* (*abstract*) – element, którego instancji nie można stworzyć. Wykorzystanie definicji wiąże się w tym przypadku z koniecznością jej doprecyzowania (relacja uogólniania). Dla tak zmodyfikowanej definicji można już powołać instancję. Element teoretyczny nie wiąże zatem bezpośrednio definicji z przebiegiem sterowania. Ma to swoje konsekwencje przy konstruowaniu sieci Petriego gdyż podsieć bezpośrednio odpowiadająca elementowi jest po zakończeniu całej konwersji usuwana. Pierwszym krokiem konwersji jest przetłumaczenie elementu teoretycznego na sieć Petriego. Jest ona później wykorzystywana przy tworzeniu elementów specjalizowanych (*specialized*) w sposób analogiczny do stosowanego dla elementów wirtualnych. Po wygenerowaniu wszystkich specjalizacji usuwa się z modelu sieć bazową.

– *statyczny* (*static*) – wszystkie instancje klasyfikatora współdzielą implementację tego elementu (używają tego samego fragmentu kodu). W sieci Petriego rozważany klasyfikator przedstawiany jest na osobnej stronie. Jego wykonanie uzależnione jest od obecności znacznika w miejscu zezwalającym (na stronie klasyfikatora). Jest on pobierany w razie wystąpienia wywołania a zwracany po wykonaniu wywołania (zasób współdzielony). Taka konstrukcja powoduje, że tylko jedna instancja może modyfikować wartości w statycznym klasyfikatorze. Zapewnia to spójność zapisywanych oraz odczytywanych danych. Dostęp do implementacji klasyfikatora statycznego zapewniają miejsca połączone fuzjami z miejscami na stronie implementacji klasyfikatora.

Wiele elementów modeli może mieć parametry. Ich definicja w UML ma postać:

```
łańcuch: String, liczba: Integer
```

w sieci Petriego wygląda ona w następujący sposób:

```
color lancuchTyp = string;
color liczbaTyp = int;
```

Ponieważ jest to definicja określająca typ znacznika, należy zadeklarować zmienne tego typu, aby móc się nimi posługiwać w wyrażeniach. Należy więc dopisać:

```
var lancuch: lancuchTyp;
var liczba: liczbaTyp;
```

Jeżeli przekazywanych ma być wiele parametrów to należy posłużyć się rekordem:

```
color zlozony = record la: lancuch * li: liczba;
```

Parametry szablonowe (template parameters) są to parametry, których typ nie jest do końca określony. W sieci Petriego typ parametru musi być całkowicie określony, więc sieć tworzy się w miejscu wywołania parametrów – gdy znany jest już ich typ.

3.4. Wspólne symbole

Niektóre symbole w UML używane są na różnego rodzaju diagramach. Zostały one opisane poniżej:

- *ramka* (*frame*) – ogranicza obszar diagramu. Ramka występuje również w sieci Petriego, ale nie jest tak restrykcyjna – można wstawiać elementy poza nią.
- *diagram tekstowy* (*text diagram*) – może zostać użyty do pokazania składni tekstowej zawartości definicji. Dla niektórych definicji jest on stosowany zwyczajowo a niekiedy może być alternatywny do graficznego przedstawienia w pozostałych diagramach. Typowym przykładem gdzie użycie diagramu tekstowego może być praktyczne jest definiowanie ciała metody. Podczas konwersji diagram tekstowy jest zastępowany przez sieć implementującą zadania w nim zawarte (jeżeli jest to możliwe).
- *symbol tekstowy* (*text symbol*) – służy do definiowania zmiennych, interfejsów, makr, typów danych itp. W sieci Petriego (w ogólności) jest reprezentowany przez deklaracje (opis w punkcie 3.3 — Ogólne konstrukcje języka).
- *symbol komentarza* (*comment*) – używany do definiowania tekstu komentarza związanego z graficznymi symbolami na diagramie. Komentarz może być również użyty w składni tekstowej. W sieci Petriego można wstawić pole tekstowe nie związane z definicją sieci. Pole to może znajdować się w prostokącie również nie związanym z definicją sieci.
- *linia komentarza* (*comment line*) – łączy symbol komentarza. W sieci Petriego można narysować linię nie związaną z wykonywaniem sieci, żeby połączyć komentarz z opisywanym elementem.

Powyższe konstrukcje mogą pomóc w orientowaniu się w sieci projektantowi diagramów UML.

3.5. Relacje w UML

Opisanie relacji pomiędzy elementami modelu w UML pozwala na przedstawienie w zwięzły sposób aspektów budowanego systemu począwszy od analizy wymagań aż po rozwiązania implementacyjne. Jest to, więc kolejny mechanizm zapewniający zbieżność tworzonego systemu ze stawianymi mu wymaganiami.

Niektóre opisane dalej relacje mogą występować na kilku różnych typach diagramów, przez co zestawienie relacji w osobnym punkcie wydaje się rozsądniejszym rozwiązaniem, niż opisywanie ich dla każdego typu diagramu.

Budując system z wykorzystaniem UML można posłużyć się relacjami:

- *zależności* (*dependency*) – relacja pomiędzy dwoma definicjami, wskazująca że jedna z nich (klient) jest zależna, z jakichś powodów od innej (dostawcy). Brak dokładnej semantyki relacji zależności powoduje jest ona użyteczna, gdy inne klasy relacji nie są odpowiednie i nie

mogą modelować danej relacji. Relację zależności można użyć w bardziej specyficzny sposób – wskazując na relację tworzenia pomiędzy instancjami klas aktywnych, to znaczy, gdy instancja używa instrukcji *new*, aby stworzyć nową instancję klasy. W takim przypadku relacja zależności może być użyta pomiędzy częściami albo pomiędzy częścią oraz symbolem zachowania, który odnosi się do maszyny stanowej w klasie aktywnej.

W sieci Petriego relacja zależności jest modelowana przez podstawione przejście na stronie klienta. Pod przejście podstawiana jest strona dostawcy. Odnosi się to również do tworzenia instancji, co zostało opisane w punkcie 6.1 —Modelowanie klas.

– *uogólnienia* (*generalization*) – relacja pomiędzy dwiema sygnaturami (na przykład klasami albo metodami) wskazującą, że jedna z nich jest bardziej ogólną sygnaturą w stosunku do drugiej. Dziedziczy ona wszystkie właściwości od bardziej ogólnej sygnatury 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 (czasami nazywany nadtypem). Oznacza to, że instancja bardziej 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.

– *realizacji* (*realization*) – jest relacją pomiędzy dwoma elementami modelu w której jeden element (klient) realizuje zachowanie, które określa inny element modelu (dostawca). Wielu klientów może realizować zachowanie pojedynczego dostawcy.

– *implementacji* (*implementation*) – szczególny rodzaj relacji realizacji. Relacja implementacji zachodzi pomiędzy klasą a interfejsem i wskazuje, że rozpatrywana klasa jest zgodna z interfejsem. Relacja ta jest wykorzystywana przy tworzeniu sieci Petriego w sposób przedstawiony w punkcie 6.1 — Modelowanie klas.

– *powiązania* (*association*) – relacja semantyczna pomiędzy dwoma lub wieloma klasyfikatorami, wskazującą, że instancje tych klasyfikatorów będą powiązane. Relacja powiązania ma dwa końce przestawione jako atrybuty. Mogą one albo oba należeć do powiązania (odzwierciedlając sytuację, gdy żaden z powiązanych klasyfikatorów nie wpływa na powiązanie) albo powiązanie może posiadać jeden z atrybutów a drugi może należeć do powiązanego z nim klasyfikatora K (odzwierciedlając sytuację, gdy powiązanie jest skierowane (grot) w stronę od K), albo każdy klasyfikator może posiadać powiązany z nim atrybut (odzwierciedlając sytuację, gdy powiązanie jest skierowane w obie strony). W przypadku, gdy powiązanie jest jednostronne drugi (zdalny) atrybut będzie istniał tylko, jeżeli jest potrzebny (na przykład, jeżeli przechowuje nazwę roli albo liczebność).

  W sieci Petriego relacja ta jest reprezentowana przez łuki związane z korzystaniem przez instancję danej klasy, z instancji innej klasy. Jeżeli połączenie nie jest skierowane, to jego

uwzględnienie w sieci zapewnia tłumaczenie innych elementów UML. Relację powiązania można skonkretyzować jako relację:

– *agregacji* (*aggregation*) – szczególny rodzaj powiązania. Łączy ono instancję klasyfikatora agregującego, znajdującą się na jednym z jej końców, z instancją klasyfikatora części znajdującą się na drugim końcu. Część agregacji może występować w więcej niż jednej relacji tego typu. Linia agregacji wskazuje, 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.
– *kompozycji* (*composition*) – szczególny rodzaj agregacji. W relacji 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 konstruktor klasy zawieranej (analogicznie destruktory). Dla czytelności sieci klas będących częściami powinny znajdować się na stronie klasy zawierającej (jako diagram struktury złożonej).

Liczebność określa jak wiele instancji może występować w relacji powiązania. Konstrukcja ta jest związana z przechowywaniem tablicy obiektów. Jej realizacja w sieci Petriego jest przedstawiona w punkcie 6.1 — Modelowanie klas.

– *rozszerzenia* (*extension*) – opisana w punkcie 4.1 — Modelowanie przypadków użycia.
– *wskazania* (*manifestation*) – szczególny rodzaj relacji zależności artefaktu od zbioru innych elementów. Wskazuje z jakich elementów zbudowany jest artefakt. Na przykład artefakt reprezentujący plik nagłówkowy w C++ może być w relacji wskazania klasy zadeklarowanej w tym pliku nagłówkowym. Ta informacja może być później użyta przez generator kodu, gdy tworzony jest fizyczny plik nagłówkowy na podstawie modelu.

  W sieci Petriego zależność ta jest modelowana przez podstawione przejścia. Zawierają one (w podstronach) definicje elementów, z których składa się artefakt. Sieć artefaktu jest tworzona przez odpowiednie połączenie przejść reprezentujących elementy. Realizuje się to według ogólnych zasad przedstawionych w niniejszej pracy.

3.6. Modelowanie pakietów

Pakiet jest podstawową konstrukcją służącą do organizowania definicji w logiczne i poręczne grupy przy modelowaniu dużych systemów. Zalecaną zasadą porządkowania jest grupowanie semantycznie bliskich (o podobnej funkcjonalności) elementów, które prawdopodobnie będą razem modyfikowane. System często składa się z wielu pakietów które zależą od siebie nawzajem co może obrazować jego architekturę.

Diagramy pakietów (*package diagram*) są wykorzystywane do przedstawiania zbiorów pakietów oraz relacji pomiędzy nimi. Stosuje się je w celu modelowania rozbicia systemu. Sieć Petriego

odpowiadająca opisywanemu diagramowi mimo, że możliwa do skonstruowania wydaje się mieć małe zastosowanie w określaniu właściwości systemu lub jego części. Przedstawiony niżej szkic algorytmu tłumaczenia ma na celu uzasadnienie powyższego poglądu.

Pakiet (*package*) jest mechanizmem organizowania elementów w logiczne grupy. Pakiet przydziela grupowanym elementom przestrzeń nazwy. Wewnątrz pakietu mogą one być adresowane bezpośrednio przy pomocy swoich nazw, natomiast z zewnątrz często jest konieczne kwalifikowanie nazw elementów modelu (`pakiet::element`).

Pakiety pozwalają także na kontrolowanie widoczności oraz praw dostępu do poszczególnych elementów w nim zdefiniowanych.

W sieci Petriego pakiet można zdefiniować jako stronę, której podstrony są elementami pakietu. Ponieważ są to definicje, poszczególne podstrony powinny być niezależne – każda jest osobną siecią. Wykorzystanie definicji zawartych w pakietach wymaga ich połączenia z innymi fragmentami sieci (analogicznie do definicji klas opisanych w kolejnym punkcie). Jak wspomniano w podrozdziale 3.3 — Ogólne konstrukcje języka, właściwości widoczności oraz praw dostępu nie będą rozpatrywane w niniejszej pracy.

Relacja sprowadzenia (*import*) jest szczególnym rodzajem zależności, która może występować pomiędzy pakietami. Może ona również prowadzić na przykład od klas lub maszyn stanowych do pakietów. Jej rolą jest sprowadzenie nazw definicji z pakietu do obecnej przestrzeni nazwy, zazwyczaj też będącej pakietem. Jest to więc środek umożliwiający opuszczenie kwalifikatorów. Nazwy definicji w pakiecie P, które zostały sprowadzone przez inny pakiet R są automatycznie dołączane do pakietów, które dalej sprowadzają lub uzyskują dostęp do pakietu R.

Dostęp (*access*) jest relacją bardzo podobną do sprowadzania. W tym przypadku zasięg widoczności ogranicza się jedynie do pakietu (R), który wykorzystuje relację dostępu. Nie jest ona przekazywana następnym pakietom (nie jest przechodnia). Z punktu widzenia architektury dostęp jest bardziej preferowany niż sprowadzanie ponieważ wymaga rozpatrzenia, które pakiety są potrzebne, a przez to nie generuje nadmiarowego „bagażu”.

Nie jest konieczne sprowadzanie albo uzyskiwanie dostępu do pakietu, aby móc odwołać się do definicji w nim zawartych. Dopóki definicja jest publiczna można się do niej odwołać przez kwalifikator. Ze względu na zrozumiałość zaleca się jednak sporządzenie opisu, jak pakiety zależą od siebie nawzajem. W sieci Petriego zależności pomiędzy pakietami (użycie) ilustruje strona hierarchii.

Pakiet nieograniczony `<<noScope>>` jest używany przeważnie, gdy występuje potrzeba podzielenia elementów pakietu pomiędzy więcej niż jednym plikiem. Jednakże może on również zostać użyty, jeżeli występuje potrzeba ustrukturyzowania zawartości pakietu w różnych częściach, ale z punktu widzenia nazw jednostek UML powinien być widoczny jako pojedyncza encja.

Semantycznie pakiet nieograniczony będzie widoczny jak każdy inny pakiet w widoku modelu i będzie również zachowywał się jak inne pakiety z uwzględnieniem przechowywania w osobnym pliku. Jednakże z semantycznego punktu widzenia wszystkie elementy w pakiecie nieograniczonym są interpretowane jako część pakietu zwierającego (w którym jest on wykorzystywany). To skutkuje na przykład, wymaganiem by `<<noScope>>` nie było używane jako część kwalifikatora odwołania

do elementu w pakiecie. Wynika z tego również, że nie jest dozwolone istnienie jednakowych nazw elementów w pakiecie nieograniczonym jak elementów w pakiecie zawierającym.

W niektórych przypadkach korzystne jest przyrostowe definiowanie pakietu jako sumy zbiorów pakietów. W zależności od pakietów, które zostały załadowane w konkretnej sesji pakiet z punktu widzenia logicznego ma różną zawartość. W TAU można to osiągnąć przy użyciu pakietów otwartych `<<openNameSpace>>`. Funkcjonuje to w następujący sposób. Definiuje się pakiety w tej samej jednostce (na przykład jako korzenie modelu). Należy nadać pakietom takie same nazwy oraz stereotypy `<<openNameSpace>>`. Z semantycznego punktu widzenia zawartość pakietów będzie teraz połączona. Oznacza to, że elementy z jednego z pakietów mogą bezpośrednio być używane w innym pakiecie bez kwalifikatora oraz, że użyte nazwy muszą być jednoznaczne w połączonych pakietach.

Możliwe jest istnienie hierarchii zagnieżdżonych pakietów otwartych, np.: nadrzędny pakiet otwarty zawierający podrzędny pakiet otwarty przechowywane w jednym pliku oraz inny plik, który również zawiera nadrzędny pakiet otwarty z podrzędnym pakietem otwartym. Jeżeli oba pliki zostaną załadowane do tego samego projektu, to zawartość pakietów nadrzędnych zostanie połączona. To samo stanie się z pakietami podrzędnymi.

Najczęstszym scenariuszem użycia pakietów otwartych, jest sytuacja gdy występuje przechowywana oddzielnie wersja podstawowa hierarchii pakietów ale zachodzi potrzeba rozszerzenia jej na przykład przez pakiet podrzędny w przypadku wykorzystania w konkretnej aplikacji.

Stereotypy `<<noScope>>` oraz `<<openNameSpace>>` pozwalające na współdzielenie są tłumaczone do sieci Petriego jako zwykłe pakiety (można je również połączyć), ponieważ odwołanie do nich jest przekształcane przy konstrukcji sieci.

4. Algorytm translacji modelu ogólnej specyfikacji systemu

Model ogólnej specyfikacji systemu jest pierwszym elementem projektowania oprogramowania obiektowego wspomaganaym przez UML. Na tym etapie wykorzystywane są informacje uzyskane podczas fazy analizy wymagań w celu przedstawienia obserwowalnych interakcji z systemem. Kolejne etapy budowy oprogramowania będą uzupełniać zaprojektowane na tym etapie szkielety algorytmów oraz architektury.

4.1. Modelowanie przypadków użycia

Modelowanie przypadków użycia skupia się na określeniu kontekstu systemu lub jego części w aspekcie aktorów którzy z nim współdziałają a także na modelowaniu wymagań zachowania tych elementów (widok systemu z zewnątrz).

Diagram przypadków użycia (*use case diagram*) przedstawia sytuację użycia przez pokazanie relacji (współdziałania) pomiędzy przypadkami użycia oraz aktorami. Jest to więc statyczny widok dynamicznych aspektów systemu. Ze względu na wysoki poziom abstrakcji diagram ten można interpretować jako stronę przedstawiającą hierarchię w sieci Petriego. Jej zawartość zależy od niższych poziomów zatem opisywanie tutaj metody jej konstrukcji wydaje się być przedwczesne.

4.2. Modelowanie scenariuszy

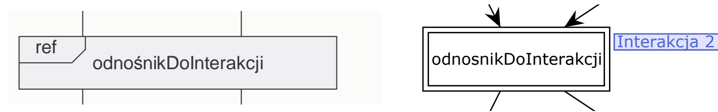
Modelowanie scenariuszy polega na opisaniu scenariuszy użycia systemu lub podsystemu. Konstruowany jest diagram sekwencji wydarzeń przedstawiający wymianę komunikatów między atrybutami. Dla skoordynowania powstałych diagramów wykorzystuje się ogólny diagram interakcji.

Interakcja jest opisem zachowania. Modelowanie interakcji polega na zapisywaniu wątków składających się z sekwencji zdarzeń. Mogą to być zdarzenia obserwowane na zewnątrz jak również związane z wykonywaniem wątku. Opisywane mogą być zarówno możliwe jak i niemożliwe scenariusze. Pozwala to na badanie zupełności modelowanego systemu.

Diagram sekwencji [1] (*sequence diagram*) określa interakcje i wymianę informacji między zbiorami atrybutów oraz inne zdarzenia. Poszczególne elementy diagramu tłumaczy się na następujące konstrukcje sieci Petriego:

[1]Nazywany jest również diagramem przebiegu.

Odnośnik do interakcji (*interaction reference*) jest mechanizmem pozwalającym na odwołanie się do osobnego diagramu (patrz rysunek 4.1). Umożliwia to zarówno ukrycie bardziej szczegółowej części scenariusza jak również wielokrotne wykorzystanie zdefiniowanych wcześniej interakcji.



Rysunek 4.1: Odnośnik do interakcji oraz jego reprezentacja w sieci Petriego

Reprezentantem w sieci Petriego jest podstawienie przejścia z odpowiednio ustalonymi miejscami wejściowymi oraz wyjściowymi. Strona, która jest podstawiana za przejście zawiera sieć odpowiadającą diagramowi, do którego odwołuje się odnośnik.

Linia życia (*lifeline*) przedstawia pojedynczego uczestnika interakcji (patrz rysunek 4.2). Jej symbol składa się z głowy oraz linii. Jeżeli linia życia nie istnieje, nie została stworzona albo została zniszczona, oznaczona jest linią przerywaną.

Rysunek 4.2: Symbol linii życia oraz jego reprezentacja w sieci Petriego

Sieć Petriego odpowiadająca konstrukcji linii życia zaczyna się od miejsca, które oznacza początek linii. Obecność znacznika w tym miejscu zależy od stanu początkowego linii — brak znacznika gdy linia ma dopiero zostać utworzona (twórca przekazuje znacznik). Jeżeli linia życia ma być aktywna od początku działania aplikacji, to w miejscu początkowym ustawia się znakowanie początkowe.

Dalszymi elementami linii życia są występujące na przemian przejścia oraz miejsca. Reprezentują one kolejne wykonywane akcje. Są częścią linii życia, ale wstawiane są tylko, jeżeli wymaga tego tłumaczona konstrukcja.

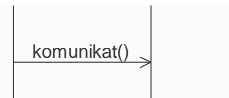
Jeżeli linia życia reprezentuje część, która występuje w wielu instancjach, poszczególne instancje muszą zostać wybrane na podstawie indeksu. Ponieważ wybór dokonywany jest na początku, to dalsze działanie związane jest tylko z jedną instancją, przez co nie wymaga zmian w tłumaczeniu sieci.

Porządek zdarzeń na linii życia jest określony przez wymianę komunikatów. Odległości pomiędzy poszczególnymi zdarzeniami nie mają proporcjonalnego znaczenia.

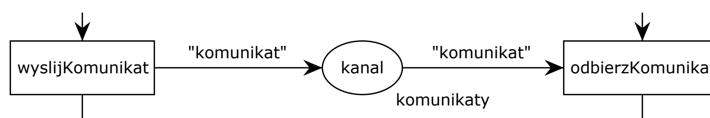
W UML za pojedynczą linię życia można podstawić fragment zawierający kilka linii (*lifeline decomposition*). Umożliwia to na przykład ukrycie komunikacji pomiędzy częściami złożonego obiektu. Istnieje oczywiście możliwość pokazania całego scenariusza (łącznie z ukrywanym fragmentem). Konstrukcja ta (powoduje pogłębienie hierarchii) jest odpowiednikiem odnośnika do inte-

rakcji (różni się orientacją), więc jej konwersja na sieć Petriego jest analogiczna (linia życia staje się podstawieniem przejścia). Może się jednak okazać, że czytelniejsza jest sieć z płytszą hierarchią.

Komunikat (*message*) jest informacją (sygnałem) przesyłaną między bytami (patrz rysunek 4.3). W normalnym przypadku składa się z dwóch zdarzeń — wysłania przez daną linię życia oraz odebrania przez inną linią życia. Komunikat może być przedstawiony jako linia pozioma albo ukośna (nie oznacza to opóźnienia). Nie może jednak dojść do sytuacji, w której zdarzenie odbioru znajduje się powyżej zdarzenia wysłania.



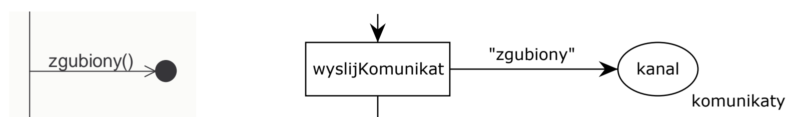
Rysunek 4.3: Symbol komunikatu



Rysunek 4.4: Reprezentacja komunikatu w sieci Petriego

Komunikat w sieci Petriego jest wysyłany przez przejście znajdujące się w linii życia nadawcy. Znacznik trafia do miejsca pomocniczego, z którego zabierany jest przez przejście znajdujące się w linii życia odbiorcy. Parametry przekazywane są w polach znacznika (w takim przypadku znacznik jest rekordem). Operacja przesłania komunikatu może być niekompletna, gdy tylko jedno z pary zdarzeń jest określone.

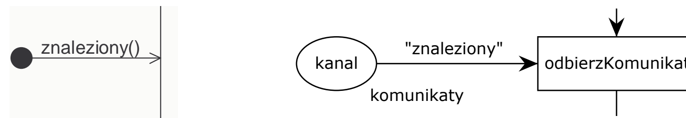
Zgubiony komunikat (*lost message*) jest komunikatem, którego zdarzenie nadawcze jest określone, ale nie istnieje zdarzenie odbiorcze (patrz rysunek 4.5). Konstrukcja może zostać użyta do opisu przypadku, gdy komunikat nigdy nie osiągnie swojego celu.



Rysunek 4.5: Symbol zgubionego komunikatu oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja jest analogiczna do tej dla zwykłego komunikatu z tą różnicą, że nie występuje przejście odbiorcy oraz związane z nim łuki.

Znaleziony komunikat (*found message*) jest komunikatem, którego zdarzenie odbiorcze jest znane, ale nie jest znane zdarzenie nadawcze (patrz rysunek 4.6). Konstrukcja może być użyta do opisu przypadku, gdy źródło komunikatu znajduje się poza modelowanym zakresem systemu. Mechanizm można użyć, aby zapobiec nadspecyfikacji — gdy kilka linii życia może być nadawcą, ale dla scenariusza nie jest istotne która.



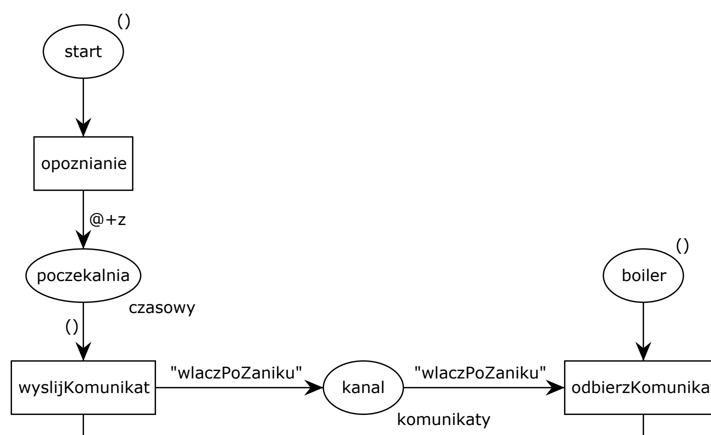
Rysunek 4.6: Symbol znalezionego komunikatu oraz jego reprezentacja w sieci Petriego

Dla sieci Petriego konstrukcja jest analogiczna do komunikatu z tą różnicą, że nie występuje przejście nadawcy oraz związane z nim łuki.

Linia określenia czasu (*time specification line*) służy do określania zależności czasowych. Może ona być jednego z 3 typów:

– Linia czasu bezwzględnego (*absolute time line*) może być połączona z linią życia tylko jednym końcem (patrz rysunek 4.7). Określa ona bezwzględny czas (`{wartość}`) albo jego zakres (`{minimum..maksimum}`).

Rysunek 4.7: Symbol linii czasu bezwzględnego



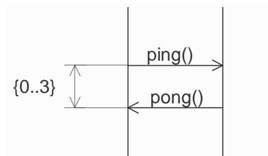
Rysunek 4.8: Reprezentacja linii czasu bezwzględnego w sieci Petriego

W sieci Petriego konstrukcja modelowana jest przez przejście oraz miejsce (patrz rysunek 4.8). W miejscu znajduje się znacznik typu czasowego. Znacznik ten jest zabierany z miejsca przez przejście które zwracając znacznik zwiększa jego pieczątkę czasową o wartość czasu bezwzględnego. Przejście znajdujące się w linii życia może zabrać znacznik gdy będzie on dostępny. Jeżeli określony jest przedział czasu bezwzględnego to należy wylosować wartość z tego przedziału. W tym celu definiuje się zmienną koloru, którego zakres jest taki sam, jak przedział czasu bezwzględnego przypisany do rozpatrywanej konstrukcji:

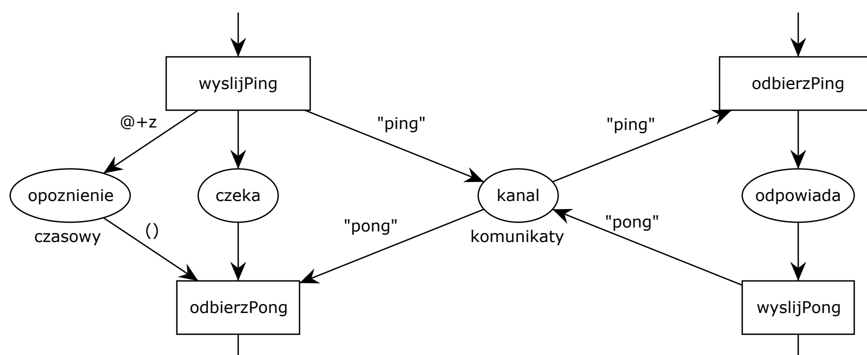
```
colset zakres=int with minimum..maksimum;
var z: zakres;
```

Należy ją wykorzystać na łuku wyjściowym przejścia należącego do linii życia. Znacznik z wylosowaną wartością jest umieszczany w miejscu wejściowym przejścia zabierającego znacznik z miejsca „opoznienie”.

– Linia czasu względnego (*relative time line*) określa czas wykonywania fragmentu linii życia (patrz rysunek 4.9). Przedział czasu można określić przez pojedynczą wartość albo jako zakres. Początek oraz koniec linii czasu względnego łączy się ze zdarzeniami na linii życia, na przykład: nadejście sygnału, wysłanie sygnału, początek – góra symbolu odnośnika, koniec — dół symbolu odnośnika.



Rysunek 4.9: Symbol linii czasu względnego

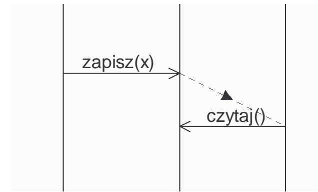


Rysunek 4.10: Reprezentacja linii czasu względnego w sieci Petriego

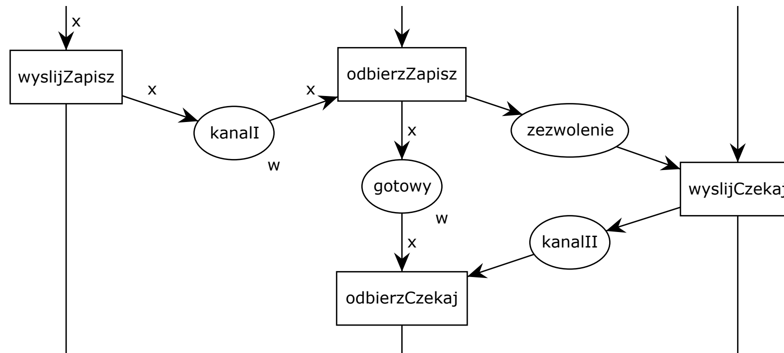
W sieci Petriego należy dodać miejsce obok „linii życia”. Gdy sterowanie wejdzie do początku linii czasu względnego przejście związane ze zdarzeniem umieści w tym miejscu znacznik z pieczątką czasową zwiększoną o wartość czasu względnego. Znacznik ten zostanie zabrany przez przejście zdarzenia końcowego linii czasu względnego.

– Linia ogólnego porządkowania (*general ordering line*) określa zależności czasowe pomiędzy dwoma liniami życia (patrz rysunek 4.11). Jest ona wykorzystywana do określenia porządku zdarzeń w różnych liniach życia bez wykorzystania komunikatów. Przeważnie nie jest konieczne określenie różnicy czasu jednak można to zrobić analogicznie jak dla linii czasu względnego.

Sieć Petriego dla tej konstrukcji jest budowana analogicznie jak dla linii czasu względnego. Różnica polega jedynie na przejściu końcowym linii które w tym przypadku znajduje się w innej „linii życia”.



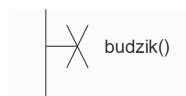
Rysunek 4.11: Symbol linii ogólnego porządkowania



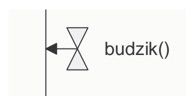
Rysunek 4.12: Reprezentacja linii ogólnego porządkowania w sieci Petriego

Budzik (*timer*) — odmierza określony okres. Budzik jest opisany przez 2 odrębne zdarzenia w interakcji. Pierwszym zdarzeniem jest ustawienie budzika (*set*) (patrz rysunek 4.13), drugim zdarzeniem jest albo zadziałanie (*timeout*) (patrz rysunek 4.14), albo wyłączenie budzika (reset) (patrz rysunek 4.15).

Rysunek 4.13: Symbol ustawienia budzika

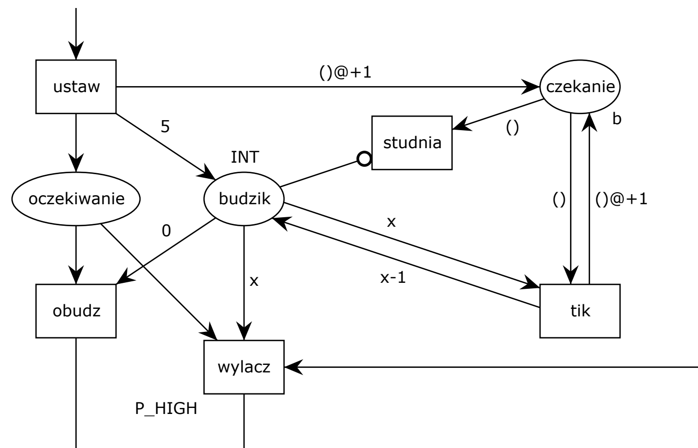


Rysunek 4.14: Symbol zadziałania budzika



Rysunek 4.15: Symbol wyłączenia budzika

Sieć Petriego dla konstrukcji budzika przedstawiona jest na rysunku 4.16. Przejście odpowiedzialne za ustawienie budzika umieszcza znacznik w miejscu reprezentującym obecny stan zegara oraz w miejscu umożliwiającym zmianę stanu zegara. Stan zegara modyfikowany jest gdy możliwe jest pobranie znacznika z miejsca „czekanie”. Jeżeli znacznik w miejscu „budzik” osiągnie wartość



Rysunek 4.16: Sieć Petriego reprezentująca funkcjonalność budzika

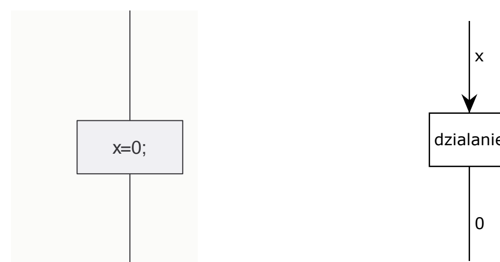
0 odpalane jest przejście „obudz", które przesyła sterowanie do dalszej części linii życia. Wyłączenie budzika możliwe jest przez odpalenie przejścia „wylacz". Zabiera ono znacznik sterowania z miejsca „oczekiwanie" oraz znacznik zegara. Powoduje to możliwość odpalenia przejścia „studnia" zabierającego znacznik z miejsca „czekanie". Taka konstrukcja zapewnia poprawną realizację budzika gdy fragment linii życia wykonywany jest wielokrotnie.

Stan (*state*) — stan w którym znajduje się byt (patrz rysunek 4.17).

Rysunek 4.17: Symbol stanu oraz jego reprezentacja w sieci Petriego

W sieci Petriego stan reprezentuje miejsce.

Działanie (*action*) — wykonanie działania (patrz rysunek 4.18).

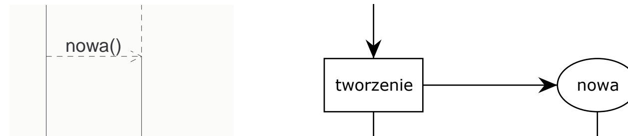


Rysunek 4.18: Symbol działania oraz jego reprezentacja w sieci Petriego

Elementowi temu odpowiada przejście. Na jego łuku wyjściowym znajduje się wyrażenie związane z wykonywanym działaniem. Jeżeli jest to skomplikowane działanie to można je zamieścić na

osobnej stronie (w postaci połączonych działań podstawowych). Stronę tą następnie podstawia się pod przejście związane z działaniem.

Tworzenie (*create*) — tworzy linię życia (patrz rysunek 4.19).



Rysunek 4.19: Symbol tworzenia linii życia oraz jego reprezentacja w sieci Petriego

W przypadku sieci Petriego twórca umieszcza w odpowiednim miejscu znacznik umożliwiając w ten sposób wykonanie podsieci. Podsieć tworzona jest według opisanych wyżej zasad.

Niszczenie (*destroy*) — kończy linię życia (patrz rysunek 4.20).

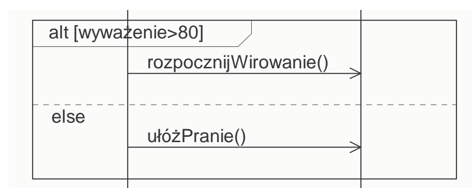
Rysunek 4.20: Symbol niszczenia linii życia oraz jego reprezentacja w sieci Petriego

W sieci Petriego jest to przejście z którego nie prowadzi ani jeden łuk wyjściowy.

Ramka wpleciona (*inline frame*) — umożliwia zgrupowanie komunikatów, które wewnątrz interakcji będą przesyłane według określonego kryterium. Pozwala to na przedstawienie złożonych konstrukcji decyzyjnych na pojedynczym diagramie. Ramka może zostać podzielona między osobne fragmenty rozgraniczone linią przerywaną.

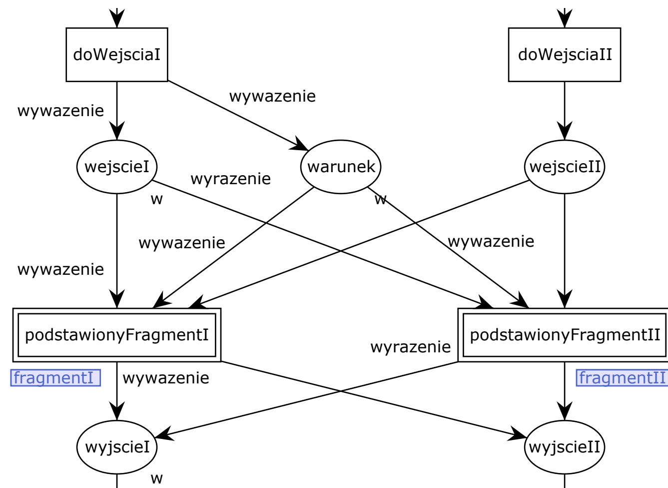
Dostępne są następujące rodzaje ramek:

– **`alt`** — wskazuje jedną z gałęzi alternatywy lub decyzji. Ramka może zostać podzielona na wiele fragmentów a każdy może być związany z warunkiem (patrz rysunek 4.21). Jedynie gałąź alternatywy, której warunek dał wynik „prawda” zostanie wybrana. Dokładnie jedna gałąź może być gałęzią **`else`**.

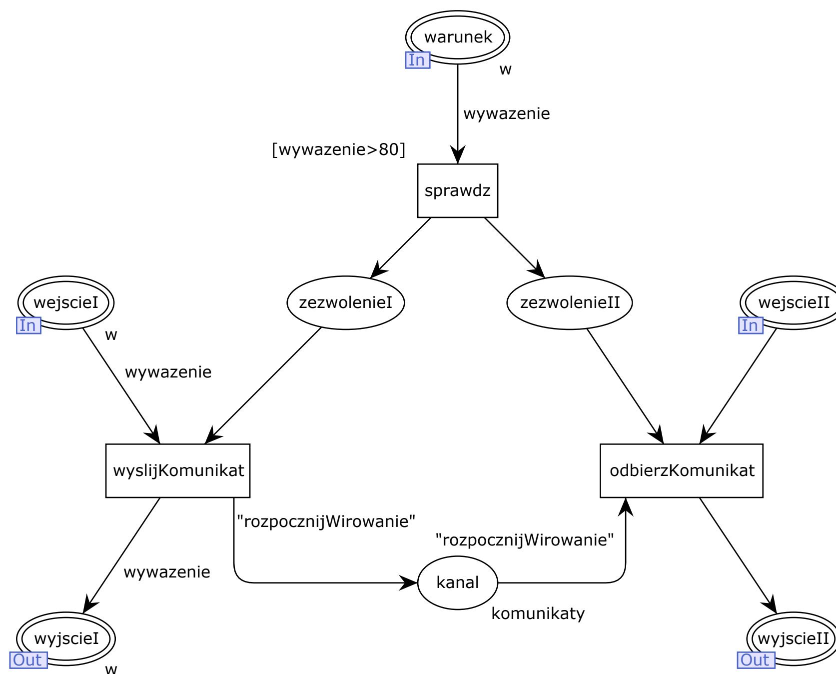


Rysunek 4.21: Symbol alternatywy

W sieci Petriego każdy z fragmentów ramki można umieścić na osobnej stronie. Rozważając samo podejmowanie decyzji (patrz rysunek 4.22): występuje wspólne miejsce, w którym



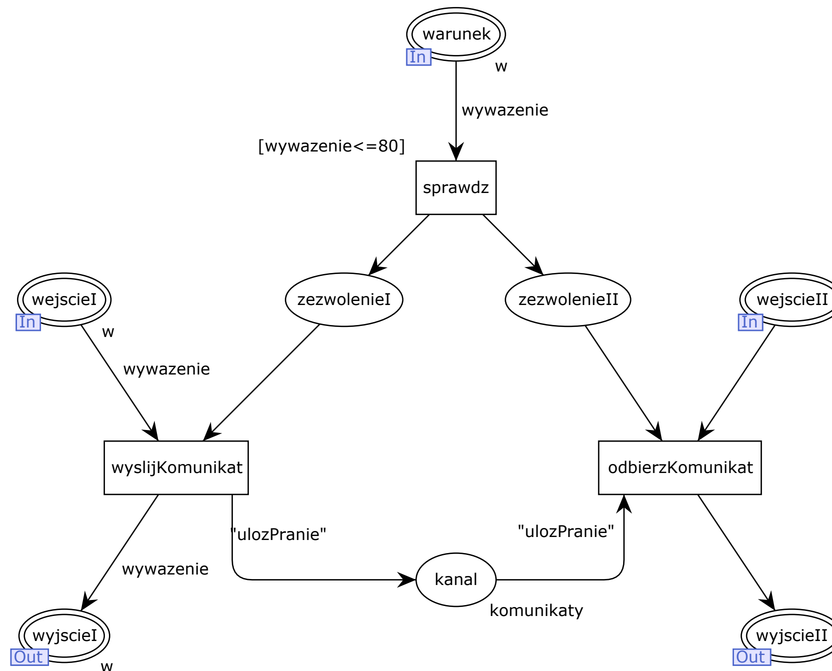
Rysunek 4.22: Nadrzędna sieć reprezentująca alternatywę



Rysunek 4.23: Sieć Petriego realizująca górny fragment alternatywy

obecność znacznika sygnalizuje możliwość wejścia do ramki wplecionej. W znaczniku zapisana jest wartość, umożliwiająca wybranie gałęzi alternatywy. Na podstronie (patrz rysunek 4.23 oraz 4.24) zawierającej tłumaczenie gałęzi znajduje się przejście z dozorem odpowiadającym warunkowi wyboru fragmentu. Spełnienie warunku powoduje wstawienie znaczników do miejsc załączających „przełączniki".

Przełącznik jest konstrukcją składającą się z przejścia oraz 2 miejsc wejściowych. Jedno z nich jest fragmentem linii życia, natomiast drugie odpowiada za dopuszczenie do przepływu sterowania w tej linii. Ilość przełączników równa jest ilości przełączanych linii życia. Na rysunkach 4.23 oraz 4.24 funkcję przełączników realizują przejścia związane z przesyłaniem

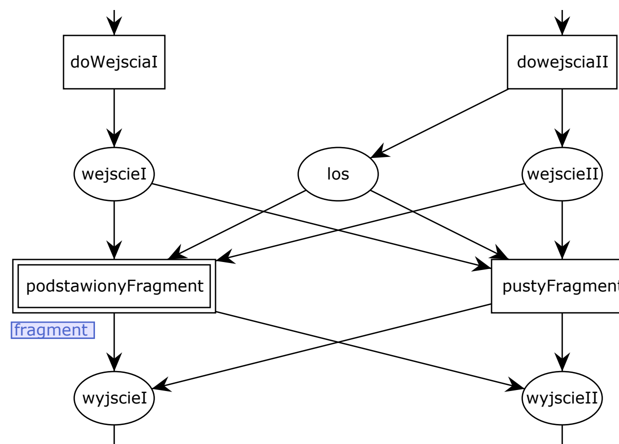


Rysunek 4.24: Sieć Petriego realizująca dolny fragment alternatywy

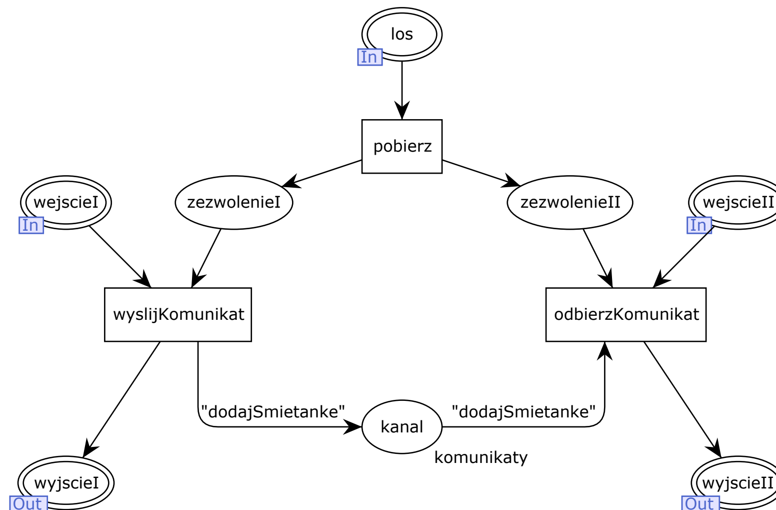
komunikatów.

– **`opt`** — wskazuje, że zgrupowane komunikaty są opcjonalne, to znaczy nie muszą się pojawić (patrz rysunek 4.25). Możliwe jest ustalenie warunku, co powoduje, że ramka zachowuje się jak alternatywa z pustą drugą gałęzią.

Rysunek 4.25: Symbol opcjonalnych komunikatów



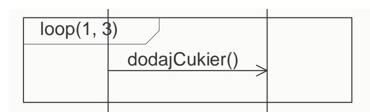
Rysunek 4.26: Nadrzędna sieć reprezentująca ramkę opcjonalną



Rysunek 4.27: Sieć Petriego realizująca ramkę opcjonalną

Sieć Petriego dla omawianej konstrukcji (patrz rysunek 4.26) jest podobna do wcześniej opisanej. Jeżeli ustalony jest warunek to przejścia mają dozory zawierające odpowiednio warunek oraz jego zaprzeczenie. W przypadku braku warunku, dozory nie występują w przejściach odpowiadających warunkom wyboru. Powoduje to losowy wybór przejścia. Pominięcie wykonania wiąże się z przesłaniem znaczników do początków linii życia za ramką.

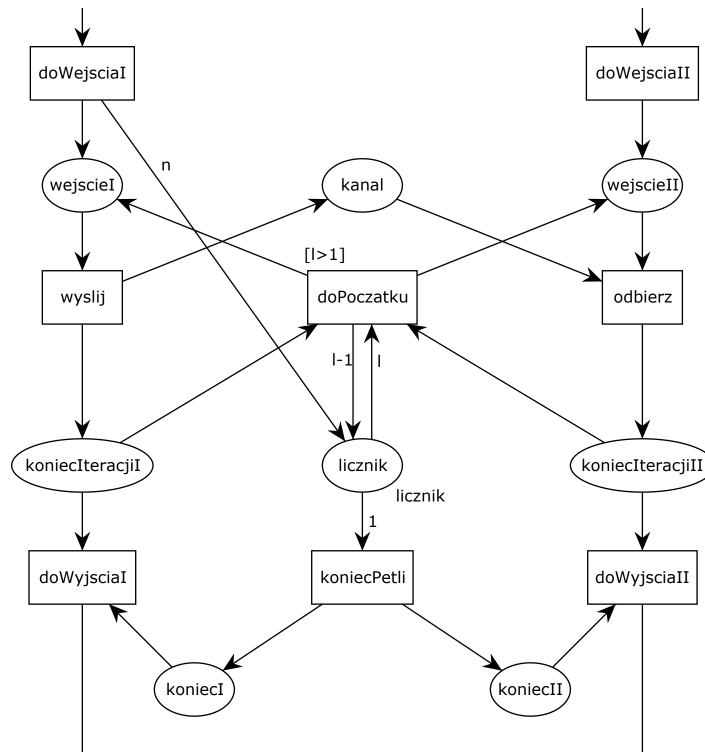
– **`loop`** — wskazuje, że zbiór komunikatów ma być powtarzany wiele razy. Ilość powtórzeń określa się jako argument wyrażenia: **`loop [`**`liczba`**`]`**. Inną możliwością jest zdefiniowanie ograniczenia dolnego oraz górnego: **`loop (`**`minimum, maksimum`**`)`** (patrz rysunek 4.28).



Rysunek 4.28: Symbol pętli

W sieci Petriego (patrz rysunek 4.29) „linie życia” mają przed końcem odpowiadającym brzegowi ramki dodatkowe przejścia (). Są one połączone z ostatnim miejscem linii życia znajdującym się w ramce. Jedno z nich odpowiada za wykonanie kolejnej iteracji — przejście do początku ramki. Drugie przejście odpowiada za wyjście z ramki wplecionej. Oba przejścia pobierają znacznik z miejsca pomocniczego. Znajduje się w nim znacznik z zapisaną liczbą iteracji. Przejście „powtarzające” zwracając znacznik dekrementuje jego wartość. Warunkiem tego przejścia jest wartość większa od 1. Przejście „kończące” wyzwala znacznik o wartości 1. Wartość początkowa jest ustalana (przed wejściem do ramki) przez przejście linii życia, która jako pierwsza wysyła sygnał w ramce. Wartość z przedziału uzyskuje się przy pomocy zmiennej, której kolor określa zakres wartości:

```
var zmienna: zakres;
```

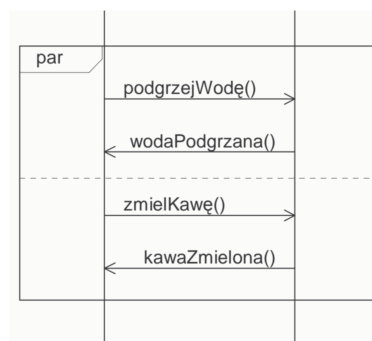


Rysunek 4.29: Sieć Petriego realizująca pętlę

gdzie

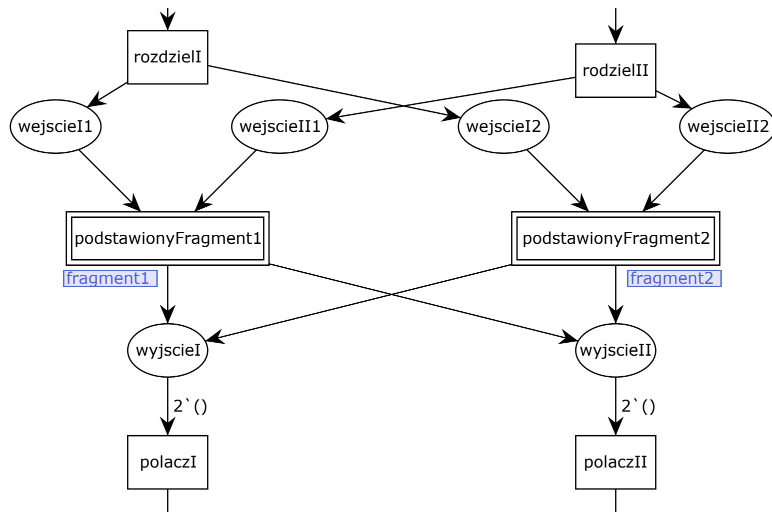
```
colset zakres=int with minimum..maksimum;
```

– **par** — wskazuje, że wiele fragmentów może się przeplatać lub występować równolegle w trakcie wykonywania (patrz rysunek 4.30). Porządek komunikatów wewnątrz fragmentów zostaje jednak zachowany.

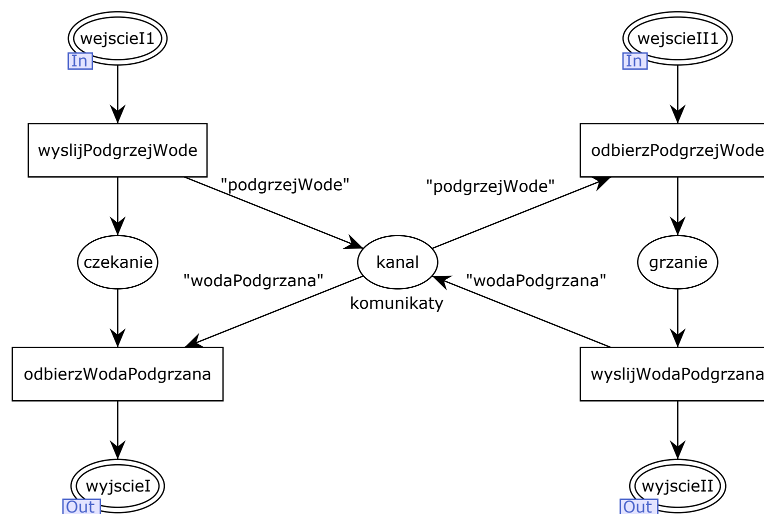


Rysunek 4.30: Symbol zrównoleglonych fragmentów

Sieć Petriego dla tej konstrukcji (patrz rysunek 4.31) składa się z przejść umieszczających znaczniki w miejscach, których liczba jest równa liczbie fragmentów ramki (dzielą linię życia). Miejsca te są miejscami wejściowymi przejść pod które podstawione są fragmenty ramki. Miejsca wyjściowe tych przejść są połączone z przejściami łączącymi rozdzielone linie życia



Rysunek 4.31: Nadrzędna sieć reprezentująca zrównoleglone fragmenty

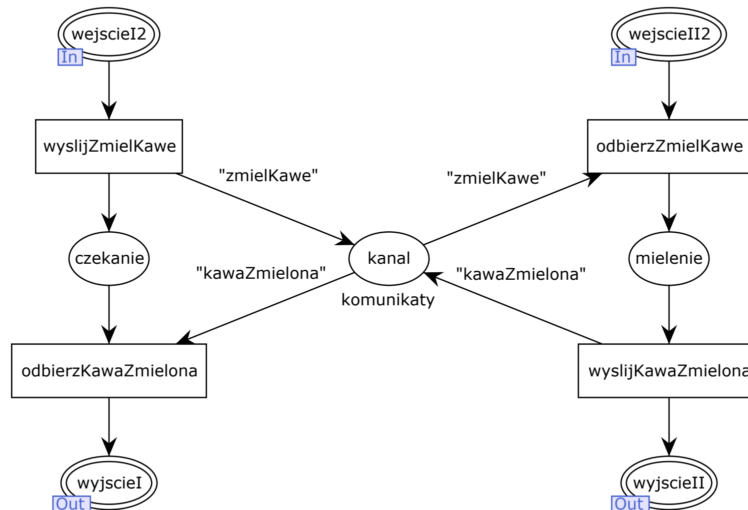


Rysunek 4.32: Sieć Petriego realizująca górną część zrównoleglenia

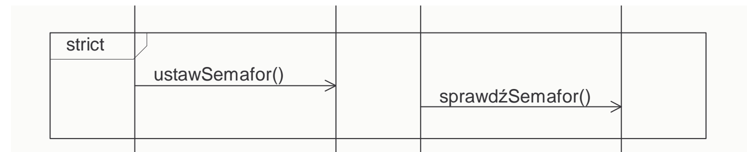
(patrz rysunek 4.32 oraz 4.33). Taka konstrukcja zapewnia opuszczenie ramki tylko w przypadku, gdy wszystkie byty zakończą działanie.

– **`strict`** — wskazuje, że komunikaty zawarte we fragmencie powinny mieć silną sekwencyjność, co oznacza, że pozioma pozycja na diagramie jest równoważna porządkowi wydarzeń (patrz rysunek 4.34). Można to zinterpretować jako istnienie wspólnego czasu dla uczestniczących linii życia.

  Sieć Petriego (rysunki przedstawione w opisie **`seq`**) dla powyższej konstrukcji wymaga stworzenia globalnego mechanizmu synchronizacji. Konieczne jest bowiem, zapewnienie zakończenia obsługi zdarzeń wcześniejszych przed rozpoczęciem obsługi działań późniejszych. Problem polega na odpowiednim zsynchronizowaniu wszystkich, objętych ramką, linii życia. Należy więc dodać konstrukcję w rodzaju linii życia, która będzie zezwalała na obsługę kolejnych zdarzeń. Składa się ona z na przemian ułożonych miejsc oraz przejść (patrz rysu-



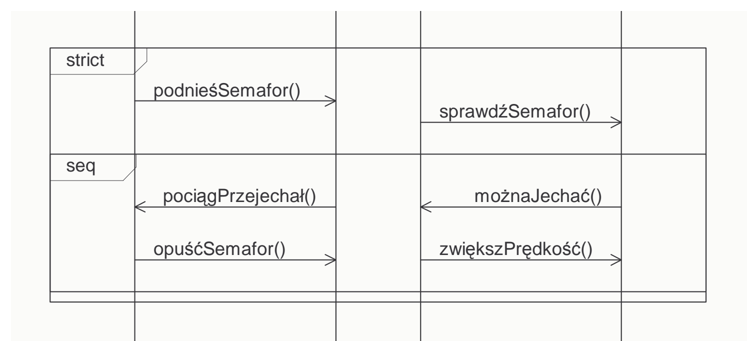
Rysunek 4.33: Sieć Petriego realizująca dolną część zrównoleglenia



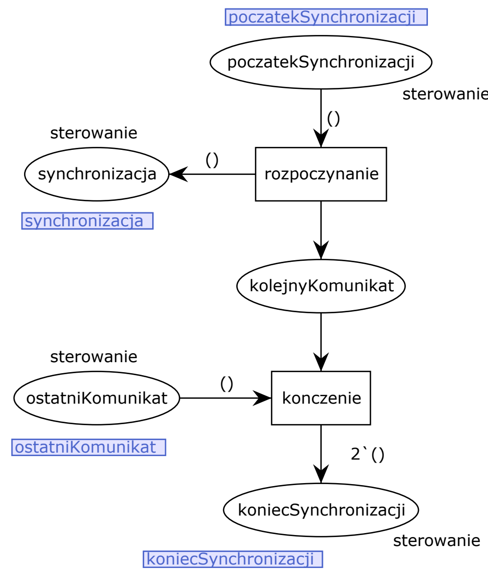
Rysunek 4.34: Symbol silnej sekwencyjności

nek 4.36). Synchronizację odpowiednich zdarzeń zapewnia przejście. Idea polega na wysyłaniu przez zdarzenie wcześniejsze znacznika do miejsca, z którego jest on zabierany przez przejście synchronizujące. Przejście to z kolei wysyła znacznik do innego miejsca, zezwalając na obsługę kolejnego zdarzenia. Jeżeli kilka zdarzeń ma się wykonać równocześnie, to przejście synchronizujące ma odpowiednio zwielokrotnioną liczbę łuków wejściowych/wyjściowych (patrz rysunek 4.37).

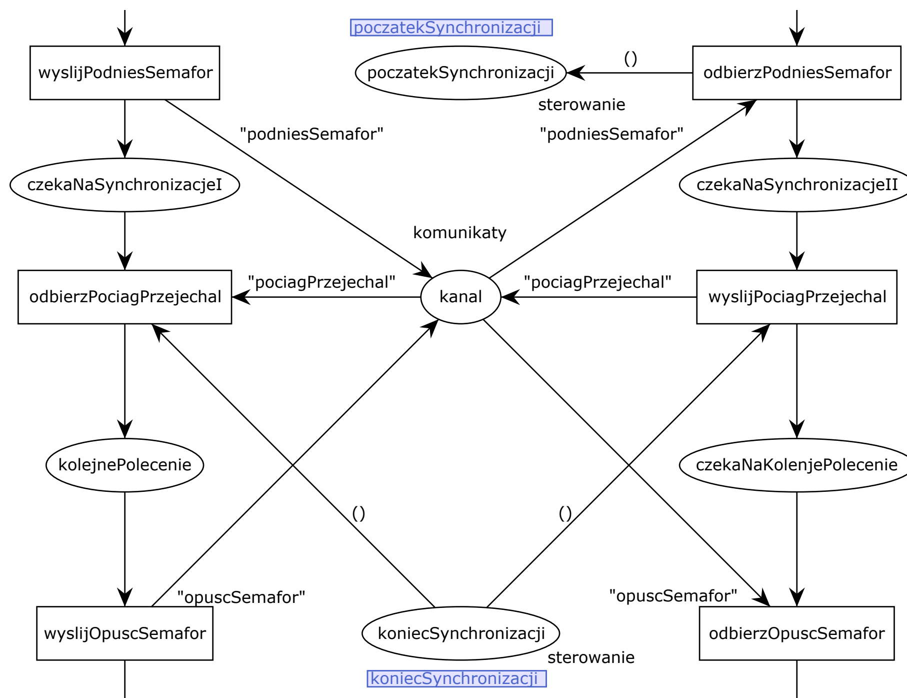
– **seq** — wskazuje na normalną semantykę diagramów sekwencji, gdzie każda linia życia jest niezależna od innych linii życia (patrz rysunek 4.35). Słaba sekwencyjność jest używana przede wszystkim do nadpisania silnej sekwencyjności.



Rysunek 4.35: Symbol słabej sekwencyjności we fragmencie silnej sekwencyjności



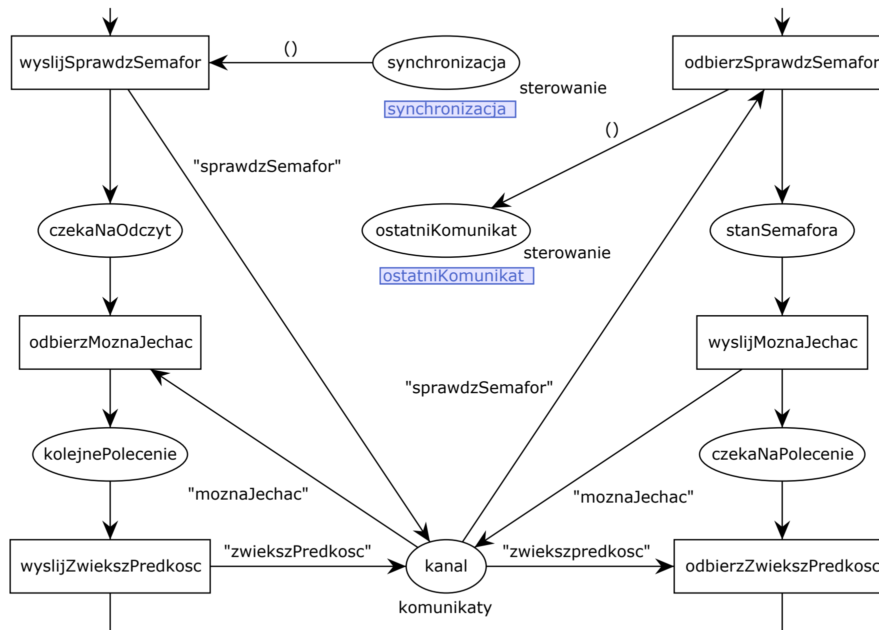
Rysunek 4.36: Sieć nadrzędna - zapewniająca synchronizację



Rysunek 4.37: Sieć Petriego dla fragmentu silnej sekwencyjności

W sieci Petriego następuje odłączenie „globalnego mechanizmu synchronizacji” (opis poniżej) od zaznaczonego fragmentu.

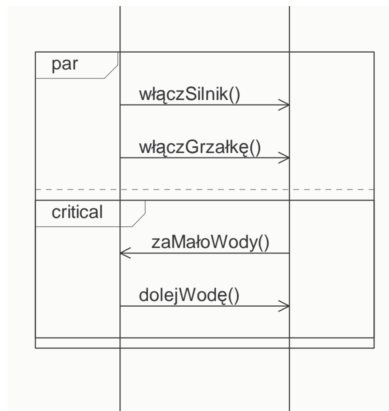
– **neg** — wskazuje, że zbiór określonych komunikatów jest niedozwolony. Jest to konstrukcja związana z fazą analizy wymagań przez co przetłumaczenie jej na sieć Petriego jest zadaniem problematycznym. Można ją wykorzystać przy sprawdzaniu właściwości sieci. Na przykład zbudować sieć dla fragmentu w ramce i wyznaczyć jego graf pokrycia. Później spróbować



Rysunek 4.38: Sieć Petriego dla fragmentu słabej sekwencyjności

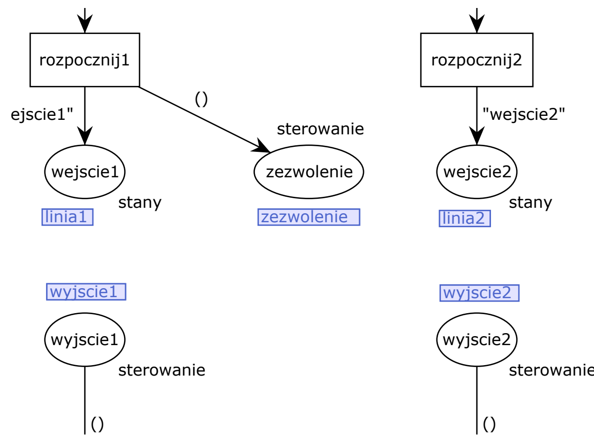
„dopasować” otrzymany graf do fragmentu grafu pokrycia całego modelu. Jeżeli jest to możliwe, oznacza to niespełnienie rozważanego wymagania.

– **`critical`** — wskazuje, że zawarte komunikaty nie mogą być przeplecione innymi ramkami wplecionymi (patrz rysunek 4.39). Można wykorzystać tę konstrukcję wewnątrz ramki równoległej do nadpisania domyślnego przeplatania zbioru komunikatów.

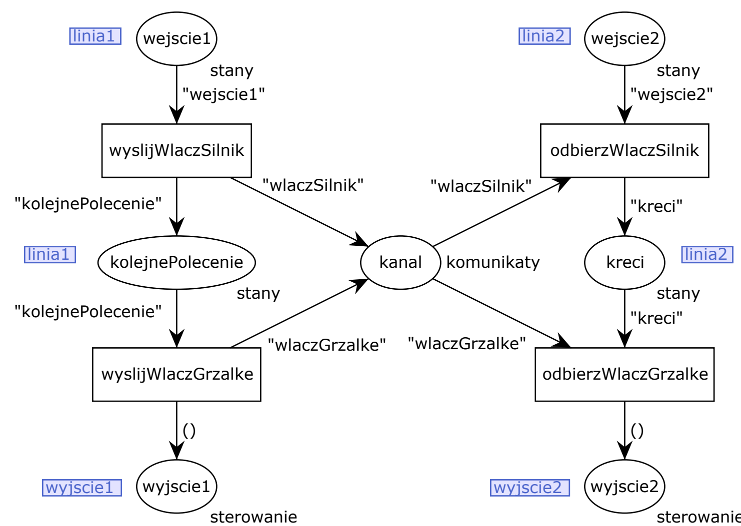


Rysunek 4.39: Symbol ramki krytycznej wewnątrz ramki równoległej

Powyższa konstrukcja w sieci Petriego jest zmodyfikowaną siecią dla instrukcji **`par`** w ten sposób, że wejście do ramki **`critical`** powoduje zabranie znaczników sterowania z pozostałych fragmentów ramki. Można to uzyskać przez połączenie miejsc we fragmentach z których może być zabrany znacznik fuzją (jedną dla każdej linii zawartej we fragmencie — rysunek 4.41). Właściwy przepływ sterowania jest w takim przypadku zapewniony przez zapis nazwy miejsca w znaczniku oraz wyrażenia na łukach. W znaczniku zapisywana jest



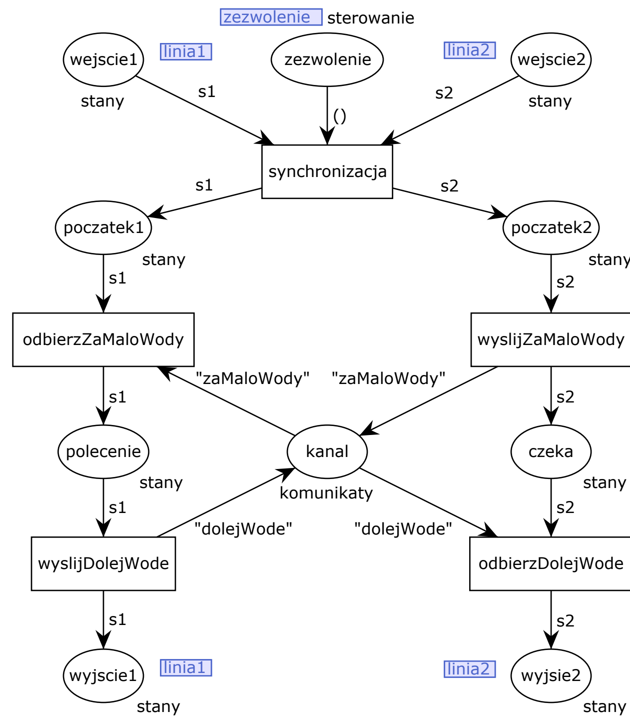
Rysunek 4.40: Sieć nadrzędna — połączenie z pozostałymi elementami diagramu



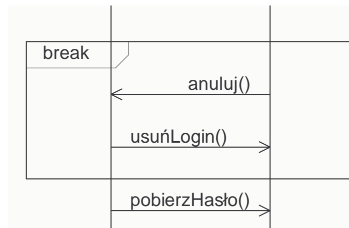
Rysunek 4.41: Sieć Petriego dla ramki równoległej

nazwa miejsca, które jest jednym z wierzchołków łuku. Łuk do przejścia ma etykietę zezwalającą na wyzwolenie przejścia tylko w przypadku, gdy jest ono następne w przepływie. Łuk wyjściowy przejścia ustawia nazwę kolejnego miejsca. Analogiczna konstrukcja jest wykorzystywana w maszynie podstanowej punkt 5.2— Modelowanie zachowania. Sieć dla ramki **`critical`** (patrz rysunek 4.42) może w dowolnej chwili pobrać znaczniki z sieci ramki **`par`**, natomiast jej wykonanie nie może zostać przerwane — żetony są przesyłane przez miejsca znajdujące się poza fuzją. Dla wyeliminowania możliwości zakleszczenia pobrane muszą być znaczniki z wszystkich linii życia (przejście „synchronizacja"). Cała konstrukcja jest dołączana do pozostałej części linii życia przy pomocy sieci przedstawionej na rysunku 4.40. Miejsce „zezwolenie" zapobiega wielokrotnemu wykonaniu ramki krytycznej.

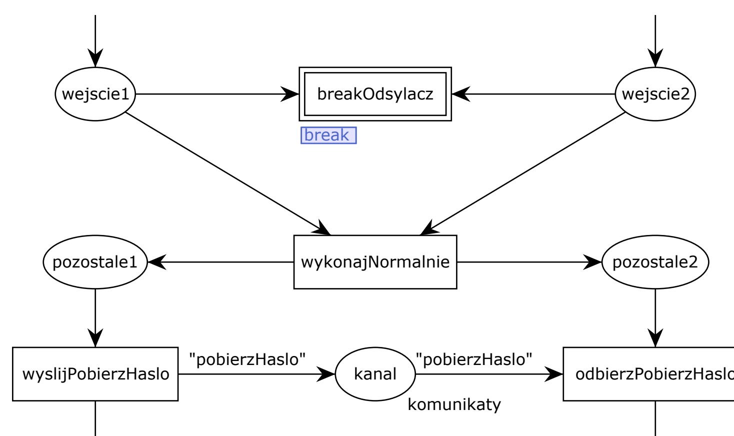
- **`break`** — wskazuje wyjątkowe zdarzenie, które przerywa pozostały diagram sekwencji a w zamian wykonuje zbiór komunikatów zawartych w ramce break (patrz rysunek 4.43).



Rysunek 4.42: Sieć Petriego dla ramki krytycznej

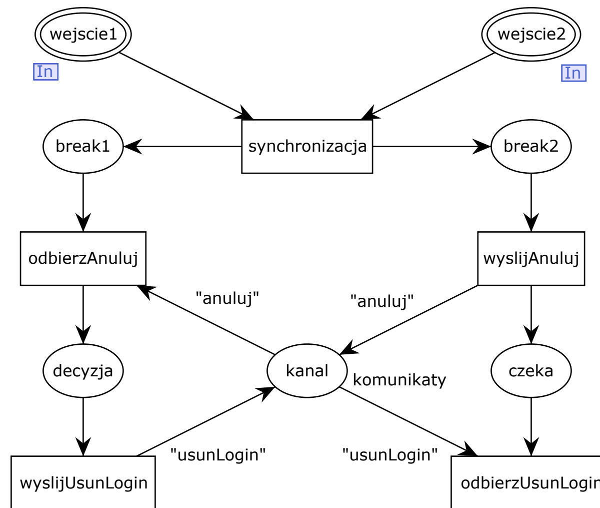


Rysunek 4.43: Symbol ramki przerywającej



Rysunek 4.44: Nadrzędna sieć reprezentująca ramkę przerywającą

Wynikowa sieć Petriego jest podobna do konstrukcji alternatywy. Można bowiem zinterpretować opisywany mechanizm jako wybór pomiędzy wykonaniem zawartości ramki i przejściem do końca linii życia, oraz pominięciem ramki i kontynuowaniem wykonywania linii życia za



Rysunek 4.45: Sieć Petriego realizująca ramkę przerywającą

ramką. Z powyższego wynika, że jedyną różnicą pomiędzy konstrukcją alternatywy oraz przerwania jest brak połączenia końców linii życia zawartości ramki z pozostałymi fragmentami linii życia diagramu.

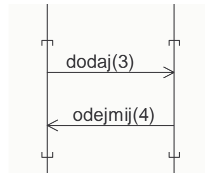
- **`assert`** — wskazuje, że sekwencje określone przez ramkę **`assert`** są jedynymi, dozwolonymi a wszystkie pozostałe sekwencje są niedozwolone. Jest to konstrukcja związana z fazą analizy wymagań, przez co nie jest bezpośrednio tłumaczona na sieć Petriego. Można ją wykorzystać w podobny sposób do przedstawionego dla instrukcji **`neg`**. Należy jedynie pamiętać o przeciwnej interpretacji wyniku.

- **`ignore`** — wskazuje, że dany zbiór komunikatów nie jest istotny i przez to nie powinien być pokazywany wewnątrz ramki. Daje to możliwość przedstawienia jedynie najbardziej znaczących komunikatów interakcji. Instrukcja ma format: **`ignore {`**`listaKomunikatów`**`}`**. Jest to konstrukcja związana z czytelnością diagramu, więc jej konwersja na sieć Petriego nie wydaje się konieczna. Można wprawdzie skonstruować odpowiednią sieć, ale przeważnie będzie ona bardziej skomplikowana niż dla przypadku nieuwzględniania ramki **`ignore`**, zwłaszcza gdy przeplatają się komunikaty ważne z nieważnymi — konstrukcja jest podobna do odnośnika do interakcji.

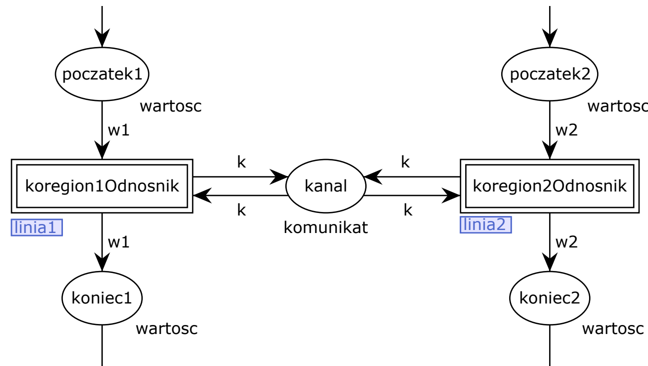
- **`consider`** — wskazuje, że dany zbiór komunikatów wewnątrz ramki jest istotny a komunikaty nie wskazane są nieważne. Instrukcja ma format: **`consider {`**`listaKomunikatów`**`}`**. Realizacja w sieci Petriego jak dla wcześniejszej konstrukcji.

Koregion (*co-region*) jest używany aby zaznaczyć, że porządek w którym komunikaty są wysyłane lub odbierane na pojedynczej linii życia nie jest ważny (patrz rysunek 4.46).

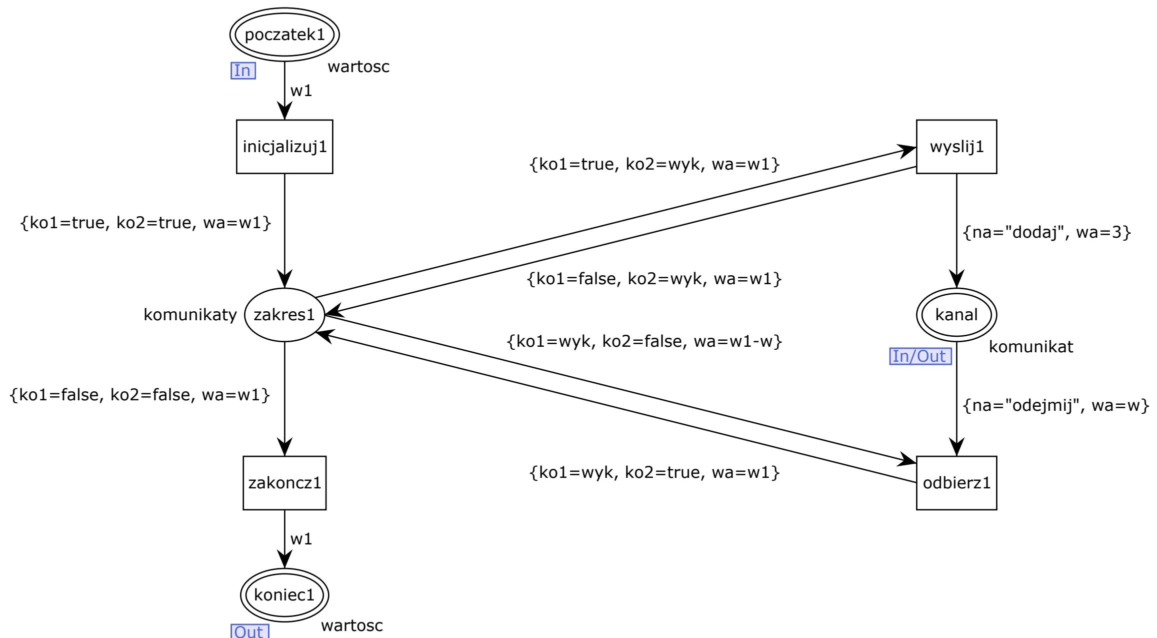
Konstrukcja wysłania komunikatów w sieci Petriego składa się w tym przypadku ze wspólnego miejsca (patrz rysunki 4.48 miejsce „zakres1” oraz 4.49 miejsce „zakres2”) do którego wstawiany



Rysunek 4.46: Symbol koregionu

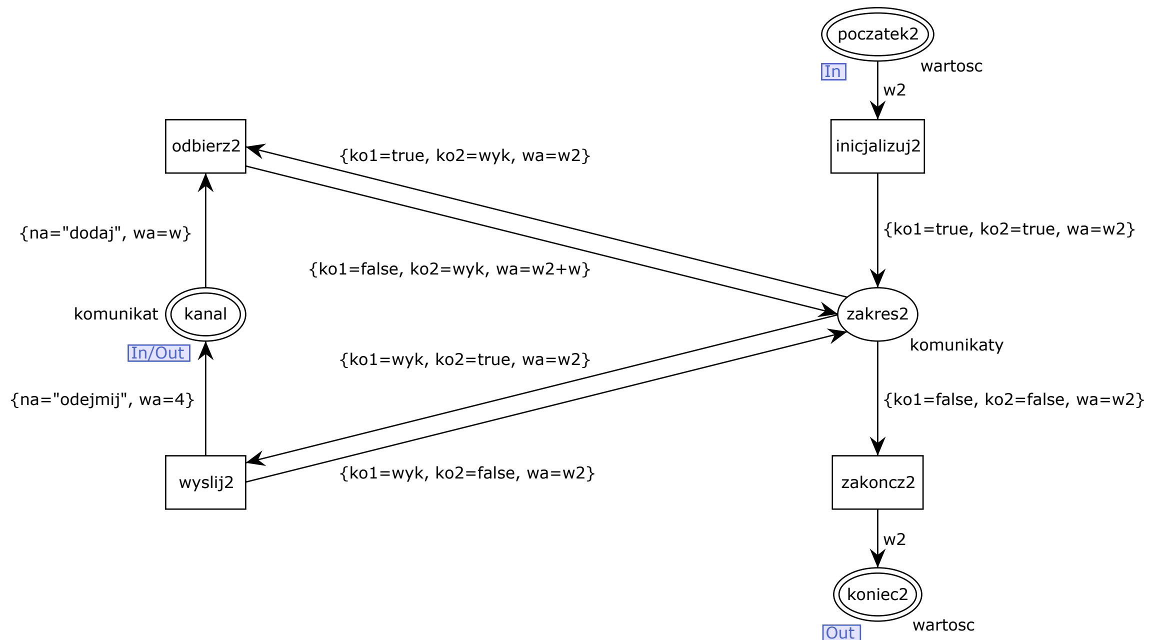


Rysunek 4.47: Nadrzędna sieć Petriego reprezentująca koregion



Rysunek 4.48: Sieć Petriego dla lewej strony koregionu

jest znacznik zawierający informację o wysłanych oraz odebranych komunikatach. Informacja ta zakodowana jest w polach binarnego rekordu, przy czym każde z nich odpowiada pojedynczemu komunikatowi. Na początku wszystkie pola zawierają wartość **`true`**. Każde przejście wysyłające komunikat może pobrać znacznik ze wspólnego miejsca jeżeli jego pole ma wartość **`true`**. W takim przypadku wysyłany jest komunikat a przejście zwracając znacznik do wspólnego miejsca ustawia wartość odpowiadającego mu pola na **`false`**. Wyjście z koregionu następuje przy pomocy przejścia, którego warunkiem wyzwolenia jest obecność na wszystkich polach rekordu wartości **`false`**.



Rysunek 4.49: Sieć Petriego dla prawej strony koregionu

Konstrukcja odbioru komunikatów jest analogiczna z tą różnicą, że przychodzący komunikat jest dodatkowym warunkiem wyzwolenia przejścia.

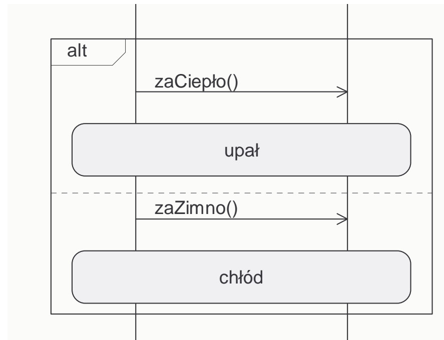
Jeżeli w koregionie występują zarówno nadawanie jak i odbiór to wykorzystywane są 2 przejścia ze wspólnym miejscem zawierającym informację o wysłanych oraz odebranych komunikatach. Jedno z przejść odpowiada za wysyłanie komunikatów, drugie za odbiór. Ich konstrukcje są analogiczne do przedstawionych wcześniej dla przypadków „jednostronnych”.

W przedstawionej konstrukcji wykorzystano typ **record** również do przesyłania danych (pole „wa”). Definicja w sieci Petriego ma następującą postać:

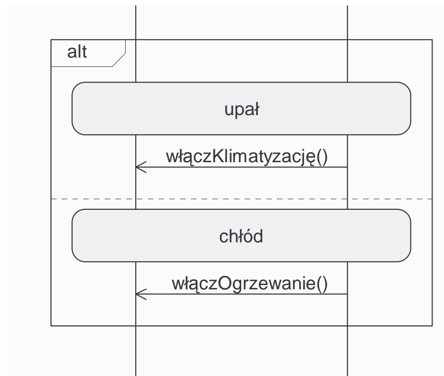
```
colset wykonany=bool;
colset nazwa=string;
colset wartosc=int;
colset komunikat=record na:nazwa*wa:wartosc;
colset komunikaty=record ko1:wykonany*ko2:wykonany*wa:wartosc;
```

Sieć nadrzędna przedstawiona jest na rysunku 4.47.

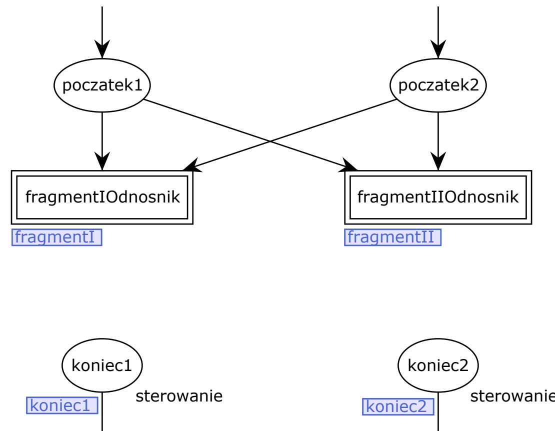
Kontynuacja (*continuation*) jest używana (jedynie w alternatywie) jak etykieta, która określa jak kontynuować daną część sekwencji. Alternatywa lub interakcja, która kończy się kontynuacją może być kontynuowana jedynie w interakcji lub alternatywie, która zaczyna się od takiej samej kontynuacji (łączy takie same etykiety). Kontynuacja jest podobna w naturze do stanu, jednak może obejmować więcej niż jedną linię życia.



Rysunek 4.50: Symbol kontynuacji — początek



Rysunek 4.51: Symbol kontynuacji — koniec

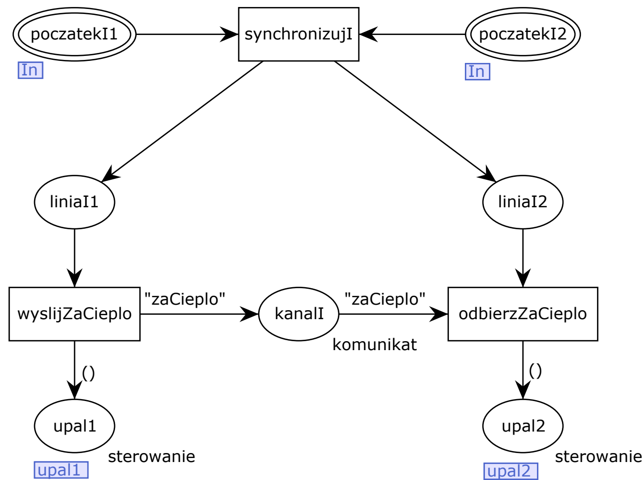


Rysunek 4.52: Sieć nadrzędna konstrukcji kontynuacji

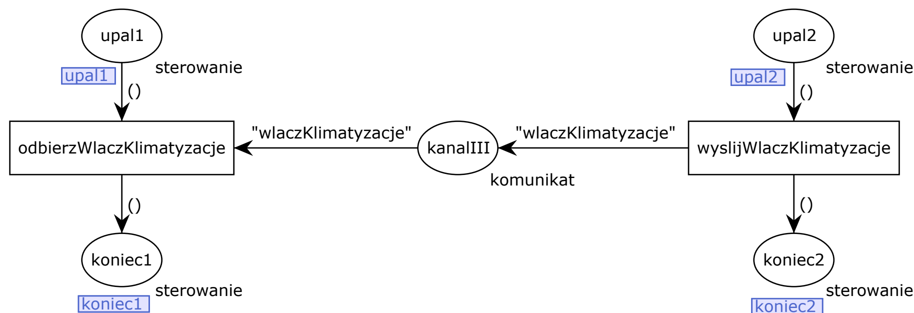
W sieci Petriego konstrukcję tę można przedstawić jako fuzję miejsc w odpowiednich liniach życia (patrz rysunek 4.52, rysunek 4.53, rysunek 4.54). Sieć Petriego dla dolnej części alternatywy przedstawia się analogicznie do górnej. Jej reprezentacja nie wniosłaby nowych informacji w związku z czym nie została umieszczona na rysunkach.

Wywołanie metody (*method call*) jest podobne do komunikatu, jednak jest zawsze synchroniczne (patrz rysunek 4.55). Oznacza to, że zawsze będzie związane z odpowiedzią metody.

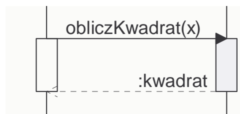
W sieci Petriego można tę konstrukcję zamodelować analogicznie do wymiany komunikatów.



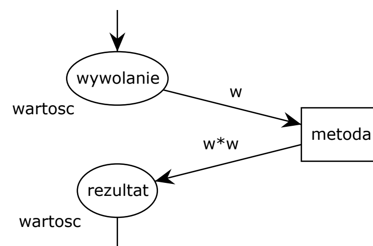
Rysunek 4.53: Sieć Petriego dla początku kontynuacji — górna część alternatywy



Rysunek 4.54: Sieć Petriego dla końca kontynuacji — górna część alternatywy



Rysunek 4.55: Symbol wywołania metody



Rysunek 4.56: Sieć Petriego dla wywołania metody

Można również podstawić stronę z metodą pod przejście na linii życia, która wykonuje metodę. Na rysunku 4.56 przedstawiono realizację metody obliczającej kwadrat przekazanej wartości. Ze względu na prostotę obliczenia wykonywane są na łuku wyjściowym przejścia — nie ma potrzeby podstawiania strony z algorytmem.

Ogólny diagram interakcji (*interaction overview diagram*) używany jest w celu zorganizowania (koordynacji) scenariuszy (diagramów) tak, aby stanowiły spójny system. Jest on rodzajem diagramu aktywności, który przedstawia przepływ sterowania pomiędzy interakcjami.

Zamiast węzłów akcji oraz węzłów obiektów używane są odnośniki do interakcji. Krawędzie aktywności oraz konstrukcje sterowania takie jak decyzja, rozwidlenie oraz koniec aktywności są takie same jak w diagramach aktywności. Konstrukcja sieci Petriego jest analogiczna jak dla diagramu aktywności. Należy jednak pamiętać o zastosowaniu odpowiednich rozwiązań dla kilku linii życia.

5. Translacja ogólnego modelu zachowania systemu

W niniejszym rozdziale zostanie przedstawiona translacja modelu zachowania systemu. Omawiane niżej diagramy mogą posłużyć zarówno do doprecyzowania algorytmów zobrazowanych przy pomocy diagramu sekwencji jak i zdefiniowania końcowej funkcjonalności aplikacji.

5.1. Modelowanie czynności

Modelowanie czynności (aktywności) polega na definiowaniu zachowania przez organizowanie go w małych jednostkach i opisywaniu przepływu sterowania oraz danych pomiędzy nimi (podobnie do diagramu przepływu). Możliwe jest również opisywanie rozmieszczenia tych jednostek w systemie.

Poniżej przedstawiono pojęcia, ich znaczenie w procesie modelowania czynności oraz translację na sieci Petriego.

Aktywność (*activity*) jest sygnaturą reprezentującą zachowanie przypadku użycia, metody lub innej encji, która może posiadać zachowanie. Aktywność dzieli zachowanie na małe jednostki (akcje) oraz steruje wykonywanienm się tych jednostek. Jest ona zazwyczaj opisywana przez diagram aktywności. W sieci Petriego konstrukcja ta reprezentowana jest przez przejście.

Implementacja aktywności (*activity implementation*) jest realizacją (zawierająca diagramy aktywności) wskazywaną przez sygnaturę aktywności. W sieci Petriego jest to strona podstawiana pod przejście reprezentujące aktywność.

Diagram aktywności (*activity diagram*) opisuje jak zachowanie jest podzielone na mniejsze fragmenty. W sieci Petriego jest to strona, na której umieszczane są elementy diagramu.

Diagramie aktywności może zawierać elementy takie, jak:

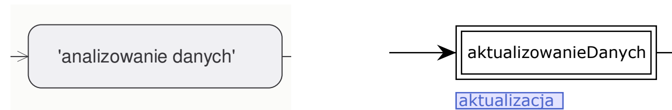
Węzeł początkowy (*initial node*) określa punkt początkowy dla przepływu sterowania (patrz rysunek 5.1). Diagram aktywności może posiadać wiele węzłów początkowych co oznacza, że przepływ sterowania zaczyna się w więcej niż jednym punkcie.

W sieci Petriego węzeł początkowy jest reprezentowany przez miejsce zawierające znacznik wstawiany przez znakowanie początkowe.

Rysunek 5.1: Symbol węzła początkowego oraz jego reprezentacja w sieci Petriego

Akcja (*action*), która jest fragmentem funkcjonalności w aktywności (patrz rysunek 5.2). Zachowanie akcji może być określone na wiele sposobów, na przykład przy pomocy operacji lub maszyny stanowej. Akcja zaczyna się wykonywać gdy zostaje osiągnięta przez przychodzący przepływ a gdy zakończy działanie przepływ jest kontynuowany od jej wyjścia.

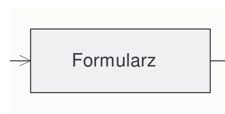
Akcji może mieć węzły złączowe przedstawiające wejściowe (argumenty) oraz wyjściowe (wyniki) parametry jej zachowania.



Rysunek 5.2: Symbol akcji oraz jego reprezentacja w sieci Petriego

W sieci Petriego akcja jest reprezentowana przez przejście pod które podstawiane jest właściwe zachowanie.

Węzeł obiektu (*object node*) reprezentuje instancję klasyfikatora (zazwyczaj klasy) uczestniczącą w przepływie (patrz rysunek 5.3). Instancja oraz jej wartości mogą być wykorzystywane w aktywności.

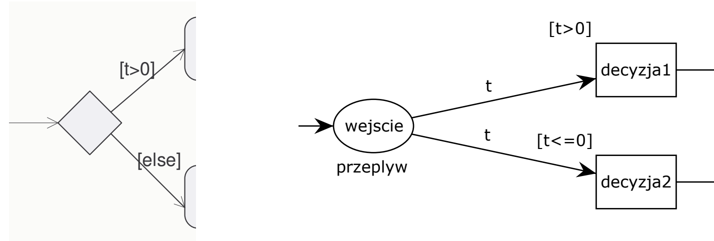


Rysunek 5.3: Symbol węzła obiektu

W sieci Petriego konstrukcja taka jest modelowana przez odwołanie do odpowiedniej strony z definicją klasy.

Decyzja (*decision*) będąca węzłem sterującym wykorzystującym warunki dozorowe (patrz rysunek 5.4). Jest ona używana w celu wyboru jednego z wielu wychodzących przepływów. Decyzja ma jeden przepływ wejściowy oraz wiele przepływów wyjściowych, każdy z dozorem. Gdy napotykany jest przepływ ze spełnionym dozorem, zostaje on wybrany.

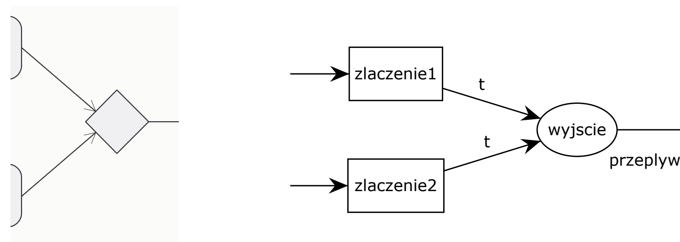
Warunki dozorów powinny zwracać wartości logiczne. Każda widoczna zmienna np. zmienna lokalna implementacji aktywności może zostać użyta w warunku dozoru. Można użyć słowa kluczowego **`else`** w dozorze żeby wskazać, że przepływ jest wybierany, jeżeli żaden z innych dozorów nie uzyskał wartości **`true`**.



Rysunek 5.4: Symbol decyzji oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja składa się z przejść mających wspólne miejsce wejściowe. Dozory tych przejść odpowiadają dozorom przepływów. Ponieważ wykorzystywane mają być zmienne muszą one zostać umieszczone w polu znacznika. Jest on więc zadeklarowany jako rekord, którego pola odpowiadają poszczególnym zmiennym używanym przez dozory. Aby konstrukcja w dowolnym przypadku działała poprawnie, każda zmienna z któregokolwiek dozoru musi mieć swoje pole w znaczniku. Wartości poszczególnych pól przypisywane są przez konstrukcje, które posługują się tymi zmiennymi (w przepływie). Następuje to na łuku wyjściowym przejścia kończącego przetwarzanie danej zmiennej (gdy wartość jest już ustalona). Jeżeli przy przepływie występuje słowo **`else`** to należy w dozorze przejścia odpowiadającego temu przepływowi wstawić warunek złożony z zaprzeczenia pozostałych dozorów.

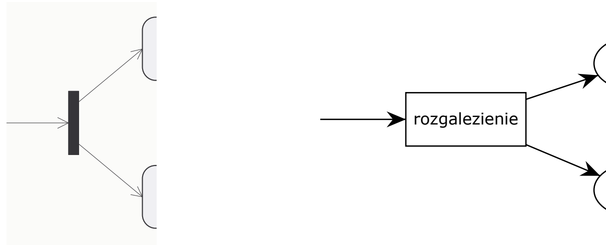
Złączenie (*merge*) jest węzłem sterującym używanym do kierowania wielu przepływów do jednego (patrz rysunek 5.5). Gdy tylko pobierany jest wejściowy przepływ, przekazywany jest wyjściowy. W przeciwieństwie do połączenia nie jest to synchronizacja przychodzących przepływów.



Rysunek 5.5: Symbol złączenia oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja ta jest reprezentowana przez przejścia które mają wspólne miejsce wyjściowe (konstrukcja odwrotna do decyzji).

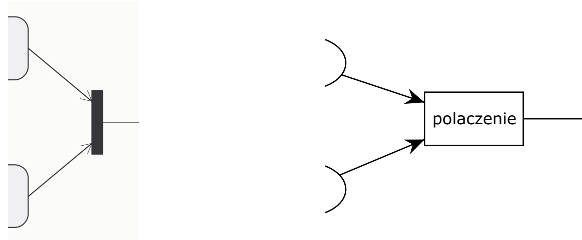
Rozgałęzienie (*fork*) jest węzłem sterującym, który rozdziela przepływ na współbieżne przepływy (patrz rysunek 5.6).



Rysunek 5.6: Symbol rozgałęzienia oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja ta jest realizowana przez przejście posiadające łuki wyjściowe, których liczba odpowiada ilości gałęzi przepływu (za rozgałęzieniem).

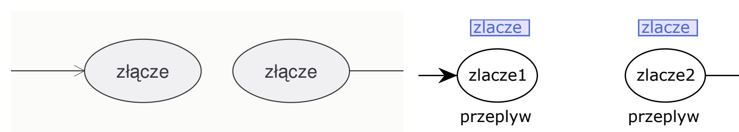
Połączenie (*join*) jest węzłem sterującym używanym do połączenia lub synchronizacji wielu przepływów współbieżnych w pojedynczy przepływ (patrz rysunek 5.7).



Rysunek 5.7: Symbol połączenia oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja ta jest realizowana przez przejście posiadające łuki wejściowe, których liczba odpowiada ilości gałęzi przepływu (przed połączeniem). Jest to konstrukcja odwrotna do rozgałęzienia.

Złącze (*connector*) jest używane jako skrót graficzny w celu uproszczenia rysowania złożonych przepływów (patrz rysunek 5.8). Krawędź aktywności może kończyć na złączu i kontynuować od innego złącza o takiej samej nazwie.

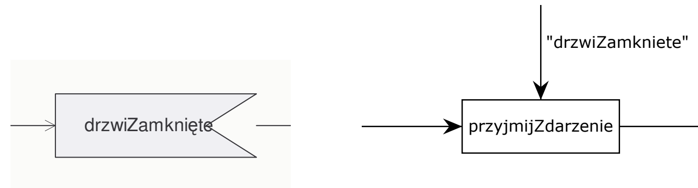


Rysunek 5.8: Symbole złącz oraz ich reprezentacja w sieci Petriego

W sieci Petriego konstrukcja ta jest modelowana przez fuzję miejsc. Złącza są reprezentowane przez miejsca. Jeżeli miejsca odpowiadają złączom o tej samej nazwie to są one połączone fuzją.

Przyjęcie zdarzenia (*accept event*) jest używane do wskazania oczekiwania na określone zdarzenie (zazwyczaj sygnał). Gdy zdarzenie zostanie otrzymane przepływ jest kontynuowany. Dane

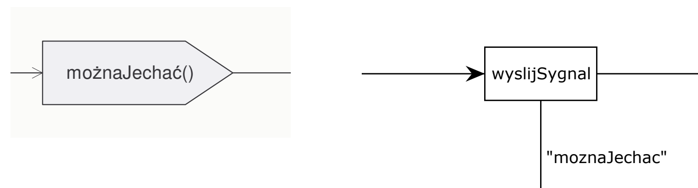
przysłane razem ze zdarzeniem mogą być użyte później w przepływie. Przyjęcie zdarzenia (patrz rysunek 5.9) jest podobne do wejścia w maszynie stanowej (opisanej w punkcie 5.2 — Modelowanie zachowania).



Rysunek 5.9: Symbol przyjęcia zdarzenia oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja ta jest modelowana przy pomocy przejścia. Ma ono 2 łuki wejściowe: 1. odpowiada za przepływ, 2. za odebranie zdarzenia. Oczekiwane zdarzenie jest znacznikiem, który może zawierać dane. Są one w takim przypadku umieszczone w polach znacznika. Jeżeli dane mają być użyte później w przepływie, to należy je zapisać (w sposób analogiczny do danych w oczekiwanym wydarzeniu), w znaczniku odpowiadającym za przepływ.

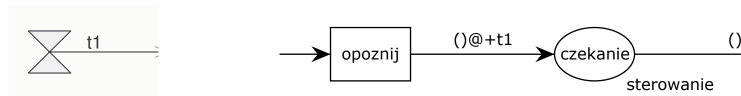
Wysłanie sygnału (*send signal*) jest realizowane przy pomocy elementu przedstawionego na rysunku 5.10. Działanie to jest podobne do akcji wyjściowej w maszynie stanowej (opisanej w punkcie 5.2 — Modelowanie zachowania).



Rysunek 5.10: Symbol wysłania sygnału oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcja składa się z przejścia, które ma 2 łuki wyjściowe. Pierwszy jest odpowiedzialny za kontynuację przepływu, natomiast drugi za wysłanie sygnału. Powoduje on umieszczenie znacznika w miejscu pośredniczącym pomiędzy wysłaniem sygnału a przyjęciem zdarzenia. Znacznik sygnału jest formatowany zgodnie z zasadami przedstawionymi w punkcie 6.1 — Modelowanie klas.

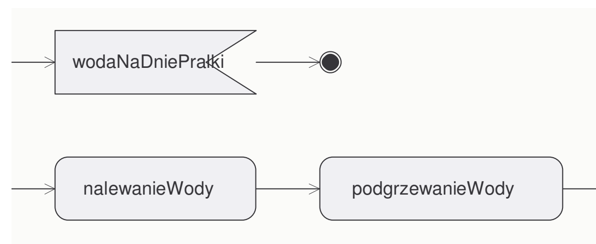
Przyjęcie zdarzenia czasowego (*accept time event*) jest używane do wskazania oczekiwania na określone zdarzenie czasowe zazwyczaj budzika, lub wartości czasu bezwzględnego (patrz rysunek 5.11). Gdy określone zdarzenie czasowe zostanie otrzymane przepływ jest kontynuowany. Konstrukcja ta jest związana z oczekiwaniem na określony moment w czasie. Może on być zdefiniowany względnie (od momentu dojścia przepływu) lub bezwzględnie (od czasu 0).



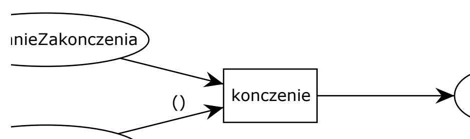
Rysunek 5.11: Symbol przyjęcia zdarzenia czasowego oraz jego reprezentacja w sieci Petriego

Jeżeli czas określony jest względnie to konstrukcja w sieci Petriego składa się z przejścia oraz miejsca. Na łuku wyjściowym przejścia znacznik otrzymuje pieczątkę czasową określającą, że może on być użyty najwcześniej po czasie zdefiniowanym w konstrukcji przyjęcia zdarzenia czasowego. Znacznik zostaje umieszczony w miejscu, w którym oczekuje na upłynięcie wyznaczonego czasu. Jeżeli czas określony jest bezwzględnie to konstrukcja w sieci Petriego składa się również z pary miejsce-przejście. W miejscu znajdującym się poza przepływem umieszczany jest (przez przejście początkowe) znacznik z pieczątką czasową ustawioną na wartość znajdującą się w oryginalnej konstrukcji. Przejście ma 2 łuki wejściowe. Jeden z nich odpowiada za zabranie znacznika z miejsca oczekiwania, natomiast drugi jest związany z przepływem. Znacznik wykorzystywany w konstrukcji czasowej musi być zadeklarowany jako `timed`.

Zakończenie aktywności (*activity final*) kończy wszystkie przepływy danej aktywności (patrz rysunek 5.12).



Rysunek 5.12: Symbol końca aktywności

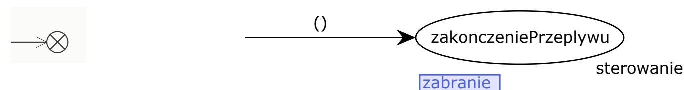


Rysunek 5.13: Sieć Petriego odpowiadająca symbolowi końca aktywności

W sieci Petriego konstrukcja ta wymaga zabrania znaczników z wszystkich przepływów aktywności. Można to osiągnąć przez połączenie miejsc pozostałych przepływów przejściami, które będą wyzwalane w razie osiągnięcia miejsca odpowiadającego końcowi aktywności. Wydaje się jednak,

że taka konstrukcja jest mało czytelna, chociaż należy ją rozważyć ze względu na rodzaj przeprowadzanej analizy sieci. Innym sposobem rozwiązania problemu jest połączenie fuzją miejsc z których znacznik może zostać zabrany. W takim przypadku należy jednak zapewnić przepływ znaczników zgodny z odwzorowywanym modelem (analogicznie do przedstawionego dalej stanu złożonego – punkt??). Można to osiągnąć przez zdefiniowanie odpowiednich wyrażeń na łukach. Na przykład ponumerować przejścia a na łuku wejściowym ustawić etykietę w taki sposób, aby znacznik mógł zostać zabrany tylko w przypadku gdy wcześniej przeszedł przez przejście mające numer mniejszy o 1. Informację którędy przeszedł ostatnio znacznik należy zapisać w osobnym jego polu, którego wartość ustawia łuk wyjściowy opuszczanego przejścia. Aby zmniejszyć zakres fuzji do minimum należałoby przeanalizować sieć pod kątem najwcześniejszego możliwego zakończenia aktywności. Może się okazać na przykład, że przepływ, który zawiera miejsce związane z końcem aktywności oczekuje wcześniej na sygnał z innego przepływu. W takim przypadku fuzja zaczynałaby się od miejsca znajdującego się za przejściem wysyłającym sygnał. Część wykonawcza końca aktywności składa się z dwóch miejsc połączonych przejściem. Pierwsze miejsce jest połączone fuzją z innymi związanymi, z opisanym wyżej rozwiązaniem. Przejście zabiera znaczniki w liczbie równej ilości kończonych przepływów i wstawia jeden znacznik do drugiego miejsca części wykonawczej.

Zakończenie przepływu (*flow final*) wskazuje na zakończenie pojedynczego przepływu w aktywności (patrz rysunek 5.14). Zakończenie dotyczy jedynie przepływu w którym zostało wstawione a nie całej aktywność. Mogą nadal występować inne przepływy działające w aktywności.



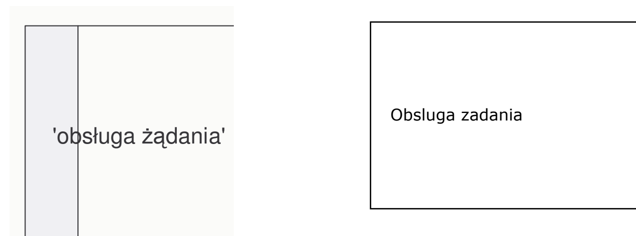
Rysunek 5.14: Symbol zakończenia przepływu oraz jego reprezentacja w sieci Petriego

W sieci Petriego konstrukcji tej odpowiada miejsce z którego znacznik może zabrać jedynie konstrukcja kończąca aktywność (połączenie przy pomocy fuzji).

Grupa (*partition*) czasami nazywaną linią przepływu, będąca mechanizmem grupującym związane ze sobą akcje (patrz rysunek 5.15). Umożliwia to podział diagramu aktywności na różne sekcje przez co ułatwia stwierdzenie, która sekcja wykonuje konkretną aktywność i jakie są przepływy danych pomiędzy poszczególnymi sekcjami.

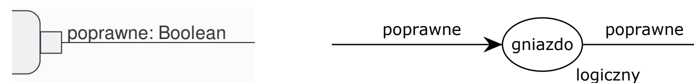
Na przykład: grupa reprezentuje wątek w systemie operacyjnym czasu rzeczywistego. Diagram pokazuje wtedy jak akcje systemu są rozdzielone na wątki

W sieci Petriego można poszczególne grupy umieścić w prostokącie nie związanym z wykonywaniem sieci (symbol graficzny).



Rysunek 5.15: Symbol grupy oraz jego reprezentacja w sieci Petriego

Gniazdo (*pin*) reprezentuje wejście albo wyjście z węzła akcji (patrz rysunek 5.16). Umożliwia przesłanie parametrów do akcji oraz pobranie wyników akcji.



Rysunek 5.16: Symbol gniazda oraz jego reprezentacja w sieci Petriego

W sieci Petriego gniazdo jest miejscem pośredniczącym w przekazywaniu parametrów. Są one przesyłane przy pomocy konstrukcji analogicznej do tej związanej z sygnałem.

Krawędź aktywności (*activity edge*) jest używana do łączenia elementów na diagramie aktywności. Oznacza to, że przepływ danych oraz sterowania może być przesyłany pomiędzy dwoma połączonymi elementami.

W sieci Petriego konstrukcja ta składa się z różnych elementów w zależności od tego co łączy. Jeżeli połączenie jest pomiędzy miejscem i przejściem to jest to łuk. Jeżeli ma łączyć miejsca to musi pomiędzy nimi wystąpić przejście. Jeżeli ma łączyć przejścia to musi pomiędzy nimi wystąpić miejsce.

5.2. Modelowanie zachowania

Model wykonywalny wymaga szczegółowego określenia zachowania metod oraz klas aktywnych.

Zachowanie w UML 2.0 można opisywać przy pomocy diagramów stanów (*statechart diagram*) albo przy pomocy opisu tekstowego (*text diagram*). W drugim przypadku model (bezstanowy) jest raczej algorytmem, którego tłumaczenie na sieci Petriego może być problematyczne.

Diagram stanów reprezentuje maszynę stanową. Może ona zostać przedstawiona na 2 sposoby: zorientowany na stany albo przejścia. Widok zorientowany na stany dobrze obrazuje złożone maszyny stanowe. Widok zorientowany na przejścia ułatwia natomiast śledzenie przepływu sterowania i komunikacji. Możliwe jest używanie obu podejść w jednym modelu.

Maszyna stanów UML jest maszyną skończenie stanową rozszerzoną o dane i obsługę sygnałów. Podstawowymi elementami maszyny stanowej są stan oraz przejście. Sterowanie znajdując

się w określonym stanie może przejść do następnego przez przejście. Akcja taka musi zostać wyzwolona przez określone zdarzenie. Można zatrzymać wykonywanie przejścia (przed osiągnięciem stanu) przez zatrzymanie całej maszyny stanowej.

Maszyna stanowa może posiadać parametry. Są one przesyłane przy wywołaniu maszyny.

Sieć odpowiadająca maszynie stanowej znajduje się na stronie, która jest podstawiana pod przejście odpowiadające symbolowi maszyny stanowej w diagramie UML. Parametry są przesyłane przy pomocy konstrukcji analogicznej do tej dla sygnałów (opisanej w punkcie 6.1 — Modelowanie klas). Jeżeli działanie maszyny może zostać zatrzymane w dowolnym momencie to należy zastosować konstrukcję analogiczną, jak dla stanu złożonego.

Na diagramie stanów mogą się znaleźć następujące konstrukcje:

Stan (*state*) — miejsce w którym zatrzymuje się sterowanie na danym poziomie hierarchii. Jeżeli jest to najniższy jej poziom, to nie jest wykonywane inne działanie. W przeciwnym przypadku sterowanie może wejść do maszyny podstanowej. Dla zwiększenia przejrzystości symbol stanu może zostać podzielony na wiele mu równoważnych. Może on również łączyć różne stany, jeżeli są początkiem tego samego przejścia. Zbiór stanów może być określony przez ich wyliczenie, lub przez wyliczenie tych, które nie należą do zbioru. Symbol `*` jest skrótem, który odnosi się do wszystkich stanów zdefiniowanych w rozważanej maszynie stanowej, poza stanami wskazanymi na liście znajdującej się za symbolem.

Jeżeli stan posiada maszynę podstanową i ma ona punkt wejściowy to może on być wskazany przy pomocy określenia **`via`**.

Rysunek 5.17: Symbol stanu oraz jego reprezentacja w sieci Petriego

W sieci Petriego stan jest reprezentowany przez miejsce. Jeżeli symbol stanu łączy kilka stanów, to stosowana jest fuzja miejsc. Jeżeli występuje dyrektywa **`via`** to właściwy punkt wejściowy określany jest przez warunki wyznaczające przebieg sterowania w stanie złożonym.

Przejście (*transition*) — sekwencja akcji, które są wykonywane gdy maszyna stanowa zmienia aktywny stan. Konstrukcja przejścia zależy od zorientowania widoku maszyny stanowej. W przypadku widoku zorientowanego na stany przejście jest reprezentowane w postaci jednego symbolu (proste przejście). Natomiast w widoku zorientowanym na przejścia, może ono (w odniesieniu do wcześniej wspomnianego pojęcia) składać się z wielu symboli. Pierwszy jest symbolem wyzwalającym po którym mogą następować symbole akcji wykonywanych przez przejście. Początkowymi symbolami przejścia mogą być: przejście wyzwalające (*triggered*) — wejście, przejście dozorowane (*guarded*) — dozór, przejście etykietowane (*labelled*) — złącze, przejście początkowe (*initial*) — start. Symbole te, różnicujące sposoby zainicjowania przejścia zostaną opisane w dalszych podpunktach.

Przejście kończy się zawsze wejściem maszyny stanowej do stanu, końca, powrotu lub przekazaniem sterowania do innego przejścia.

Następny stan z historii (*history nextstate*) — kieruje do stanu, który wyzwolił przejście. Konstrukcja może zostać użyta w obu rodzajach widoków.

Rysunek 5.18: Symbol przekierowania do stanu odwiedzonego wcześniej stanu

W sieci Petriego konstrukję tą można zrealizować przez przejście odsyłające sterowanie do odpowiedniego miejsca.

Historia może być głęboka (*deep*) umożliwiając cofanie się do podstanów (rekursywnie) składających się na stan poprzedni (w maszynie podstanowej).

W sieci Petriego należy wówczas przesłać (użyć jako znacznik sterowania) do konstrukcji związanej z głęboką historią znacznik maszyny podstanowej. Zostanie on później odesłany umożliwiając kontynuowanie wykonywania maszyny podstanowej od miejsca, z którego został zabrany. Pozostała część konstrukcji jest taka sama jak we wcześniejszym przypadku.

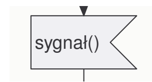
Ponieważ stan może łączyć wiele stanów (na przykład przy pomocy $\star$) więc w takim przypadku historia (płytka) ma wracać do ostatniego rzeczywistego stanu. Wiąże się to z koniecznością uzupełnienia fuzji o mechanizmy pamiętania (w znaczniku) i wyboru (na łukach) analogiczne jak dla maszyny podstanowej.

Wejście (*input*) — określa wyzwalacz przejścia. Zazwyczaj jest to sygnał, ale może być również budzik albo metoda. Jeżeli to samo zachowanie przejścia w danym stanie ma być wywoływane przez różne wyzwalacze to można użyć listy identyfikatorów w symbolu wejściowym. UML umożliwia wyzwolenie wejścia przez dowolny sygnał przy pomocy symbolu $\star$. Sygnał może posiadać parametry co pozwala na przesyłanie wartości.

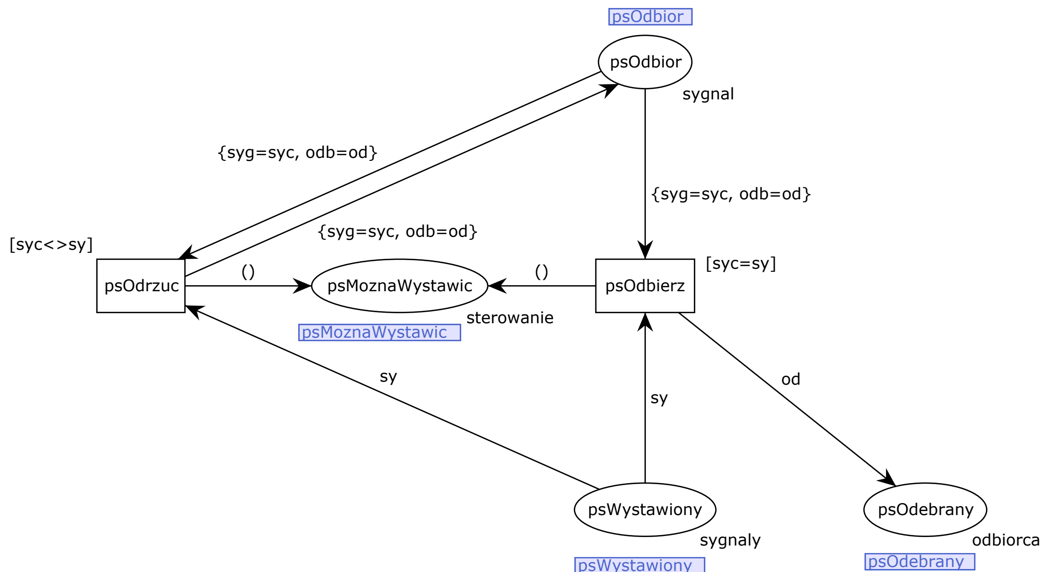
Z maszyną stanową związana jest kolejka przechowująca przychodzące sygnały — według kolejności nadejścia.

Obsługa sygnałów w UML wiąże się też z następującymi warunkami:

- jeżeli nie ma żyjącej instancji maszyny stanowej na końcu ścieżki komunikacji — sygnał zostanie utracony,
- jeżeli cel odnosi się do maszyny stanowej, która zakończyła działanie — sygnał zostanie utracony,
- jeżeli odbierająca maszyna stanowa jest w stanie, w którym nie może obsłużyć sygnału — zostanie on utracony.



Rysunek 5.19: Symbol wejścia



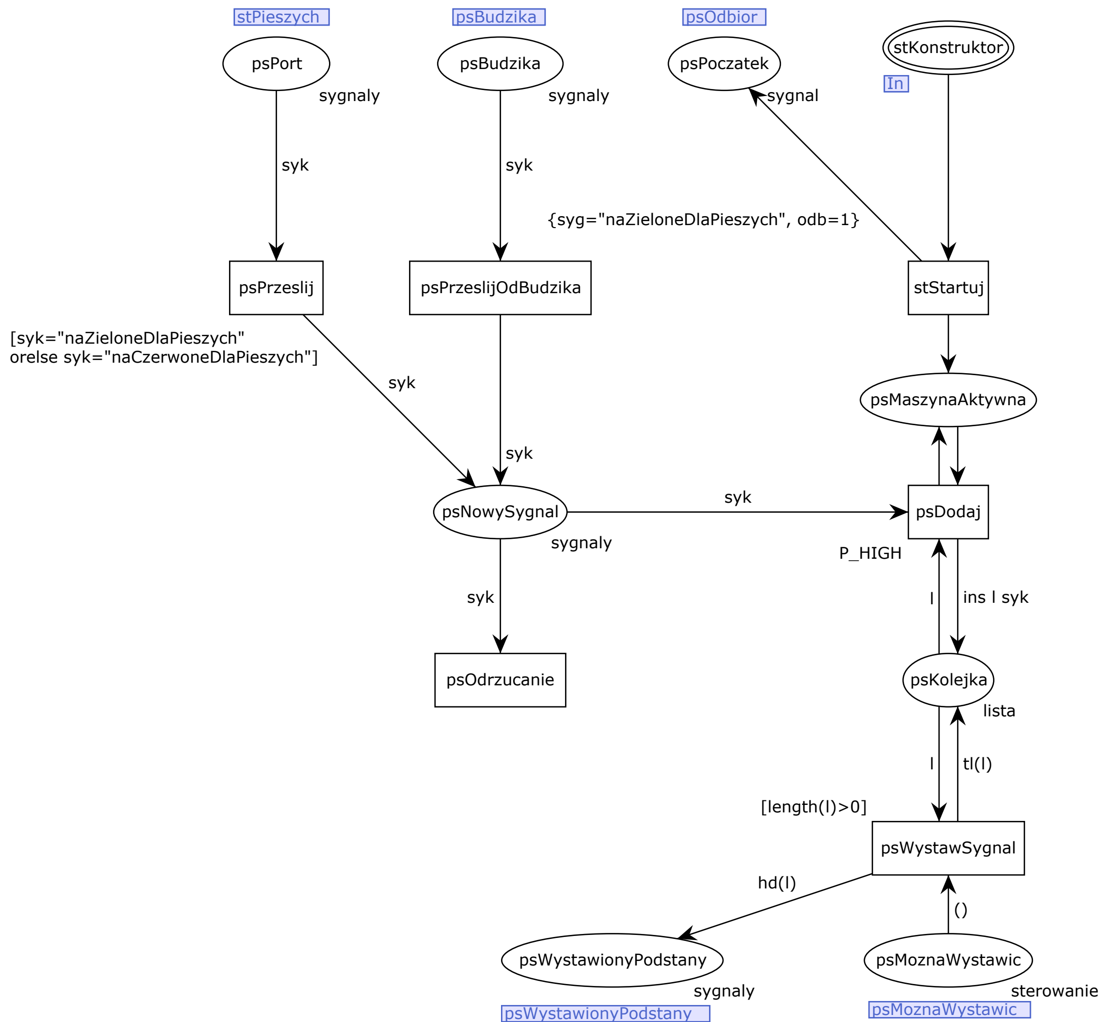
Rysunek 5.20: Konstrukcja reprezentująca odbieranie sygnału w sieci Petriego

W sieci Petriego przychodzący sygnał jest dołączany do listy (patrz rysunek 5.21). Jest ona umieszczona w znaczniku natomiast typ jej elementów zdefiniowany jest analogicznie do listy sygnałów (elementami są rekordy). Znacznik ten jest pobierany z miejsca przechowywania przez konstrukcję, która dołącza nadchodzący sygnał i zwraca znacznik z listą.

Fragment ten składa się więc z przejścia które pobiera znacznik kolejki. Jest ono wyzwalane nadejściem nowego sygnału. Przejście to zwraca listę dołączając nowy sygnał (`ins lista sygnal`).

Jeżeli jest to możliwe znacznik z listą jest pobierany a jego pierwszy element (`hd(lista)`) jest umieszczany w kolejnym miejscu jako sygnał. Przejście udostępniające kolejny sygnał może zostać odpalone jeżeli lista nie jest pusta. Jest to sprawdzane przez określenie długości listy (`length(lista)>0`). Zwracana jest lista bez pierwszego (udostępnionego) elementu (`tl(lista)`).

Konstrukcja odbioru znajduje się na osobnej stronie (patrz rysunek 5.20). Znacznik sygnału pobierany jest z miejsca połączonego fuzją z miejscem wystawienia sygnału w konstrukcji kolejki. Rozważany znacznik może zostać pobrany przez jedno z 2 przejść. Pierwsze odpowiada za odrzucenie sygnału. Jest ono wykonywane gdy maszyna zgłosi, że nie oczekuje sygnału — pusta nazwa oczekiwanego sygnału. Drugie przejście odpowiada za odbiór sygnału. Jest ono wyzwalane, gdy nazwa sygnału oczekiwanego zgadza się z nazwą sygnału wystawionego. Wyrażenia podejmujące decyzje znajdują się w dozorach przejść. Porównują one nazwę przychodzącego sygnału z wymaganą `#nazwa(sygnal)=wymaganaNazwa`, gdzie `sygnal` jest zmienną przesyłającą sygnał.



Rysunek 5.21: Konstrukcja reprezentująca kolejkę sygnałów w sieci Petriego

Jeżeli dowolny sygnał może wyzwolić przejście to kryterium wyboru jest zawsze prawdziwe (można je pominąć) dla przejścia odbierającego i zawsze fałszywe dla przejścia odrzucającego niezgodny sygnał (można pominąć przejście). Przejście „odbierające" zezwala na wystawienie kolejnego sygnału. Miejsce „odbior" jest połączone fuzją z miejscem w którym maszyna określa jakiego sygnału oczekuje. Jest to zapisane w znaczniku w postaci nazwy sygnału. Jeżeli ten sam sygnał może być odebrany w różnych miejscach maszyny należy zdefiniować znacznik jako rekord, którego pierwsze pole opisuje nazwę sygnału, natomiast drugie identyfikuje konstrukcję uprawnioną do odbioru. Informacja ta jest przekazywana do miejsca oznaczającego odbiór. Jest ono połączone fuzją z miejscem oznaczającym kontynuację przepływu sterowania w maszynie stanowej. Łuk wyjściowy tego miejsca może zawierać warunek odnośnie do wymaganego numeru odbiorcy gdy ten sam sygnał może być odebrany w wielu miejscach. Można dla potrzeb analizy umieszczać odrzucone sygnały w osobnym miejscu.

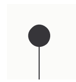
Przy konstruowaniu sieci trzeba zwrócić uwagę na zapewnienie unikalnych nazw fuzji dla różnych kolejek sygnałów. Jeżeli występuje więcej niż jedna instancja maszyny stanowej, należy zapewnić ich rozróżnialność. Uzyskuje się to w sposób analogiczny do rozróżnienia obiektów jednej

klasy.

Ponieważ nieczynna maszyna stanowa nie powinna odbierać sygnałów, należy dodać przejście, które pobierze znacznik sygnału gdy nie może on zostać dołączony do kolejki. Aby zapewnić determinizm przejście dołączające sygnał musi mieć wyższy priorytet.

Sygnał zostanie pobrany, jeżeli istnieje instancja do której jest skierowany.W przeciwnym przypadku sygnał zostanie zabrany przez przejście odpowiedzialne za jego odrzucenie.

Początek (*start*) — określa punkt początkowy maszyny stanowej albo podstanowej. Wyznacza on więc przejście początkowe wykonywane zaraz po utworzeniu maszyny. W maszynie podstanowej początek jest nazywany wejściowym punktem łączącym (*entry connection point*) ponieważ umożliwia wejście do stanu złożonego. Wejściowe punkty łączące odnoszą się również do symbolu stanu następnego jako wejścia do stanu złożonego.



Rysunek 5.22: Symbol początku

W sieci Petriego początek jest miejscem od którego zaczyna się sieć. Jest ono różne od pierwszego stanu maszyny (pod)stanowej, ponieważ następujące po nim przejście związane jest z obsługą maszyny (włączenie kolejki sygnałów) oraz ewentualnych dodatkowych działań.

W maszynie podstanowej miejsce początkowe połączone jest fuzją z miejscem nadrzędnym. Przejście początkowe może zostać odpalone przez znacznik maszyny nadrzędnej. Odpowiedni warunek umieszczony jest na łuku wejściowym przejścia (stan poprzedni → następny – maszyna podstanowa). Jeżeli początek nie ma nazwy, to przy tworzeniu sieci należy mu ją nadać.

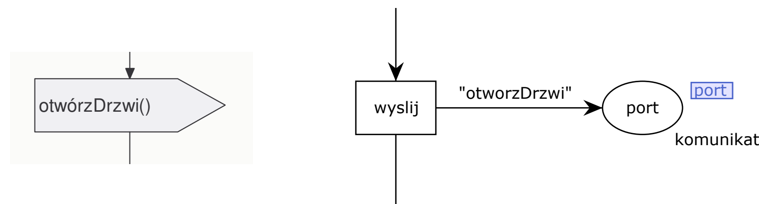
Działanie (*action*) — zazwyczaj używane w symbolu zadania. Określa się je przy pomocy składni tekstowej. Dostępne są następujące działania:

definicja zmiennej (`typ nazwa`)
puste wyrażenie (**`;`**)
złożone działanie (**`{}`**)
przypisanie (**`=`**)
działanie: wyjście (**`output`**), nowy (**`new`**),
ustawienie (**`set`**), wymazanie (**`reset`**)
wyrażenie wyjątku (**`try/catch`**)
wyrażenie warunkowe (**`if`**)
wyrażenie rozstrzygające (**`switch`**)
wyrażenie kodu docelowego (**`[[]]`**)
wyrażenie dopóki (**`while`**)
wyrażenie dla (**`for`**)
wyrażenie kasujące (**`delete`**)

wyrażenie kończące: powrót (**`return`**), przerwanie (**`break`**), kontynuacja (**`continue`**), zatrzymanie (**`stop`**), następny stan (**`nextstate`**), idź do (**`goto`**)

Nie wszystkie działania zostaną dokładnie opisane. Niektóre, ze względu na podobieństwo do innych konstrukcji będą pominięte. Kilka z powyższych działań ma też składnię graficzną przez przypisane odpowiednie symbole.

Wyjście (*output*) — umożliwia (przejściu) wysłanie sygnału do innej maszyny stanowej, otoczenia, lub wewnątrz tej samej maszyny stanowej. Sygnał może mieć parametry, które nie muszą być ustawione przy wysłaniu. Można umieścić więcej niż jeden sygnał na wyjściu co zostanie zinterpretowane jako osobne, następujące po sobie wyjścia.



Rysunek 5.23: Symbol wyjścia oraz jego reprezentacja w sieci Petriego

W sieci Petriego wysłanie sygnału reprezentowane jest przez łuk wyjściowy z przejścia. Budowa sygnału jest analogiczna do opisanej w punkcie 6.1 — Modelowanie klas. W przypadku wielu sygnałów liczba łuków „wysyłających” sygnały zależy od ilości różnych maszyn stanowych, do których są one wysyłane. Jeżeli ma być wysłanych kilka sygnałów do tej samej maszyny to można je wysłać jako wielozbiór (**++**).

Adresowanie

Adres jest zapisywany w znaczniku w postaci rekordu zawierającego 2 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 na 2. 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. UML udostępnia wiele sposobów adresowania. Zostaną one przedstawione poniżej wraz z reprezentacją w sieci Petriego.

Bezpośrednie adresowanie sygnału jest wyrażane przy pomocy kropki — `odbiorca.sygnał1`. W tym przypadku sygnał jest odwołaniem się do cechy odbiorcy (fragmentu jego realizacji). W sieci Petriego bezpośrednie adresowanie polega na określeniu łącz-

nika (punkt 6.2 — Diagram Struktury Złożonej) oraz numeru obiektu. Sygnał zostanie umieszczony w kolejce wskazanego obiektu.

Jeżeli nie jest określony adres lub ścieżka sygnał zostanie wysłany na jedną z możliwych ścieżek. W sieci Petriego adres odbiorcy wygląda następująco: `{nazwaInstancji=" ", numerObiektu= 1}`.

Wyrażenie **`this`** odnoszące się do instancji, w której jest użyte. Jeżeli jest to metoda klasy pasywnej odnosi się do instancji klasy pasywnej. Jeżeli jest używane w metodzie klasy aktywnej albo jej maszynie stanowej odnosi się do instancji klasy aktywnej. W sieci Petriego jest to adres wyznaczony zgodnie z przedstawionymi wyżej zasadami. W tym przypadku adres odbiorcy będzie taki sam jak adres nadawcy.

Jeżeli jest wskazany identyfikator portu, sygnał zostanie wysłany przez ten port. Jeżeli dana klasa ma zdefiniowany anonimowy port, który realizuje dokładnie jeden interfejs, identyfikator może być też nazwą interfejsu. W tym przypadku odnosi się on do anonimowego portu. W sieci Petriego znacznik wysyłany jest przez odpowiednie miejsce reprezentujące wskazany port. W praktyce polega to na połączeniu przejścia wysyłającego sygnał łukiem z określonym miejscem.

Jako cel sygnału mogą być podane zmienna albo atrybut. Muszą one być typu interfejs (wyjście przez — **`via`**) albo klasą aktywną (albo specjalnym typem *Pid*). W sieci Petriego sygnałowi nadawany jest odpowiedni adres zgodnie z wyżej przedstawionymi zasadami.

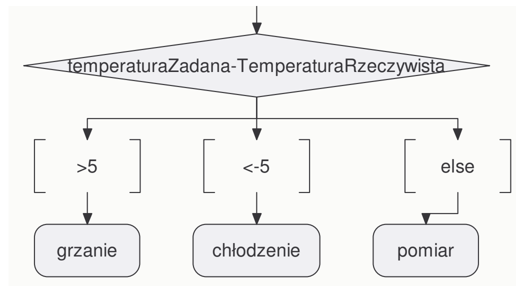
Atrybut może się również odnosić do wartości pośrednich: **`self`**, **`sender`**, **`parent`**, **`offspring`** opisanych w podpunkcie dotyczącym typu *Pid*. Wartości adresów wymaganych w tym przypadku są pobierane z atrybutów instancji. Szczegóły konstrukcji przedstawione są w podpunkcie dotyczącym wyrażeń. Sygnał można wysłać do wyrażenia. Musi ono być typu interfejs albo klasa aktywna (albo *Pid*). Jest to konstrukcja podobna do opisanej powyżej. Różnica polega na tym, że można użyć bardziej złożonego wyrażenia w parametrach. Na przykład wyłuskanie pola albo łańcucha. Sieć Petriego dla tej konstrukcji jest analogiczna do wyżej przedstawionej. Wyrażenie jest umieszczone na łuku wyjściowym przejścia wysyłającego sygnał.

Rozstrzyganie (*decision*) – używane w przejściu do wykonywania alternatywnych działań (mechanizm typu **`switch`**). Składa się ono z części pytającej oraz części odpowiadających. Część pytająca zawiera wyrażenie rozwiązywane na początku wykonywania rozstrzygania. Część odpowiadająca zawiera wyrażenia zakresu prowadzące do różnych częściowych przejść. Ścieżka zostanie wybrana jeżeli jej etykieta będzie się zgadzać z wynikiem wyrażenia.

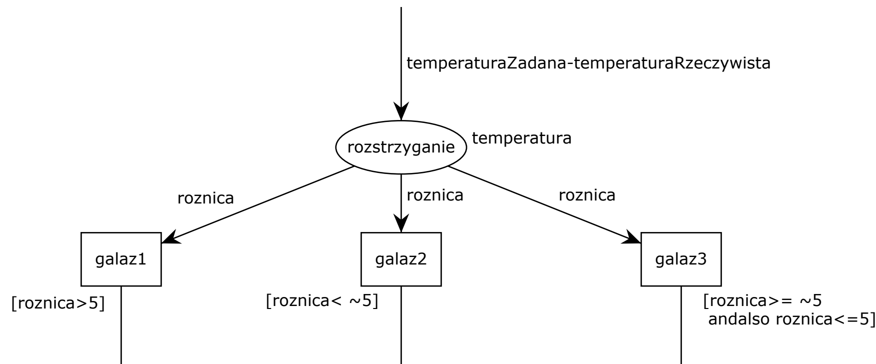
Sieć Petriego składa się odpowiednio z przejścia na którego łuku wyjściowym wykonywane jest wyrażenie (patrz rysunek 5.25). Wartość zwracana zapisywana jest w znaczniku, który umieszczany jest w miejscu wyjściowym przejścia. Z tego miejsca znacznik zabierany jest przez przejście, którego dozór odpowiada wyrażeniu w odpowiedniej gałęzi maszyny stanowej.

Wyrażenie zakresu może być zdefiniowane przez:

- określoną wartość — w dozorze porównanie (`zmienna=wartość`) wartości ze znacznika (zmienna) z ustaloną



Rysunek 5.24: Symbol rozstrzygania



Rysunek 5.25: Reprezentacja rozstrzygania w sieci Petriego

– otwarty zakres — w dozorze sprawdzenie relacji wartości ze znacznika z ustaloną; relacje zapisywane są w takim sam sposób z wyjątkiem nierówności, która w UML oznaczana jest przez **!=** a w sieci Petriego **<>**.

– zamknięty zakres (`..`) — w dozorze sprawdzenie zawierania się wartości pomiędzy dwoma granicami: `zmienna`**>**`dolnaGranica` **`andalso`** `zmienna`**<**`gornaGranica` (zmienna — wartość ze znacznika)

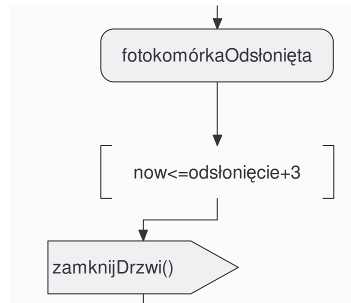
– listę oddzielonych przecinkami wyżej przedstawionych możliwości — w dozorze połączone instrukcją **`orelse`** wyżej przedstawione możliwości.

Jest możliwe opisanie rozstrzygania niedeterministycznego. Uzyskuje się to przy pomocy wyrażenia **`any`** i pozostawienie pustych odpowiedzi.

W sieci Petriego wybór niedeterministyczny modeluje się przy pomocy przejść, które pobierają znacznik z jednego miejsca. Gdy nie ma żadnych warunków to które przejście zostanie odpalone jest decyzją losową.

Dozór (*guard*) — kieruje przebiegiem sterowania (może, ale nie musi posiadać wyzwalacz). Dozór może sprawdzać wyrażenie i w zależności od jego wyniku zezwalać albo nie na dalsze wykonanie. Wyzwolenie przejścia może nastąpić tylko w przypadku gdy warunek zwróci wartość **`true`**. Inną ewentualnością jest wykorzystanie dozoru do wyboru punktu wyjściowego maszyny podstanowej stanu źródłowego przejścia. Jest ono wykonywane gdy sterowanie wyjdzie z maszyny podstanowej przez punkt określony w dozorze. Jeżeli dozór posiada wyzwalacz sprawdzenie warunku

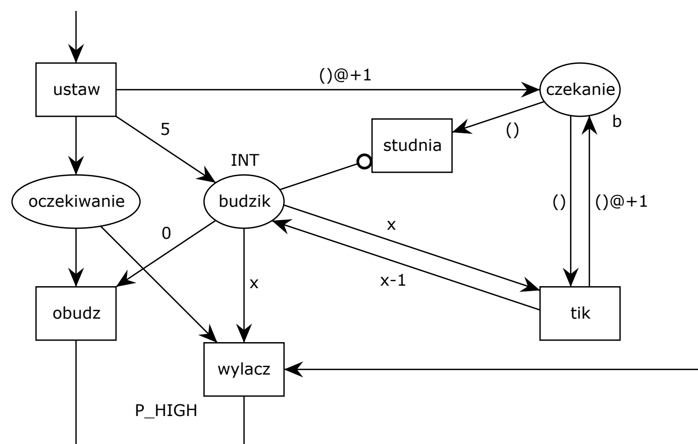
wykonywane jest po zdarzeniu wyzwalającym. Jeżeli wyrażenie zwróci wartość prawdziwą (**`true`**) przejście zostanie odpalone. Jeżeli wynikiem będzie fałsz (**`false`**) maszyna stanowa pozostanie w stanie a sygnał, który spowodował wyzwolenie zostanie zachowany w kolejce sygnałów.



Rysunek 5.26: Symbol dozoru

Sieć Petriego dla tej konstrukcji składa się z przejścia, którego dozór odpowiada wyrażeniu w dozorze maszyny stanowej. Jeżeli dozór ma sprawdzać punkt wyjścia to informacja o nim znajduje się w znaczniku (pole rekordu). Zostanie to opisane w podpunkcie dotyczącym stanu złożonego. Jeżeli dozór posiada wyzwalacz to jego konstrukcja składa się z 2 przejść. Jedno z nich odpowiada za sytuację gdy warunek jest spełniony. Pobiera ono sygnał wyzwalający określony na łuku wejściowym. Warunek dozoru umieszczony jest w dozorze przejścia. Wznawia ono przepływ sterowania oraz umieszcza znacznik zezwalający na wystawienie kolejnego sygnału. Drugie z przejść odpowiada za sytuację niespełnienia warunku dozoru. Ma ono taką samą etykietę na łuku wejściowym oraz przeciwny warunek w dozorze. Nie zezwala ono na przepływ sterowania, umieszcza sygnał, który je wyzwolił w kolejce (zapis sygnału) oraz zezwala na wystawienie nowego sygnału.

Ustawienie budzika (*timer set*) — tworzy i aktywuje instancję budzika. Powtórne ustawienie aktywnej instancji budzika wyłącza (niejawnie) poprzednią i tworzy nową instancję budzika.



Rysunek 5.27: Sieć Petriego reprezentująca funkcjonalność budzika

Sieć Petriego dla konstrukcji budzika przedstawiona jest na rysunku 5.27. Przejście odpowiedzialne za ustawienie budzika umieszcza znacznik w miejscu reprezentującym obecny stan zegara

oraz w miejscu umożliwiającym zmianę stanu zegara. Stan zegara modyfikowany jest gdy możliwe jest pobranie znacznika z miejsca „czekanie". Jeżeli znacznik w miejscu „budzik" osiągnie wartość 0 odpalane jest przejście „obudz", które przesyła sterowanie do dalszej części linii życia. Wyłączenie budzika możliwe jest przez odpalenie przejścia „wylacz". Zabiera ono znacznik sterowania z miejsca „oczekiwanie" oraz znacznik zegara. Powoduje to możliwość odpalenia przejścia „studnia" zabierającego znacznik z miejsca „czekanie". Taka konstrukcja zapewnia poprawną realizację budzika gdy fragment linii życia wykonywany jest wielokrotnie.

Ponieważ budzik może być przestawiany należy wykorzystać konstrukcję sprawdzającą czy jest on aktywny (jej opis znajduje się w dalszej części punktu). Ustawienie budzika może odbywać się na 2 sposoby. Są one reprezentowane przez 2 przejścia prowadzące od miejsca początkowego konstrukcji budzika. Pierwsze z przejść odpowiada za ustawienie nieaktywnego budzika. Może ono zostać odpalone pod warunkiem (na łuku), że budzik jest nieaktywny. Drugie może zostać odpalone pod warunkiem, że budzik jest aktywny. Pobiera ono znacznik aktywnego budzika i wstawia znacznik nowego budzika.

Wyłączanie budzika (*timer reset*) — wyłącza budzik jeżeli jest aktywny. Sieć Petriego realizująca to zadanie została opisana we wcześniejszym punkcie.

Zadanie (*task*) — zachowanie przejścia zapisane tekstowo.

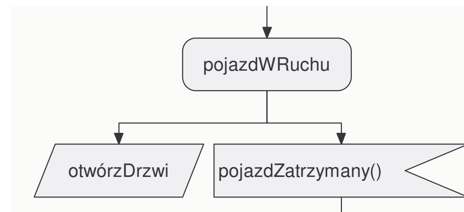
W sieci Petriego konstrukcja ta będzie przedstawiona w zależności od zawartości zadania. Należy się jednak liczyć z koniecznością rozpisania poszczególnych instrukcji na osobne elementy sieci. Konstrukcje związane z niektórymi instrukcjami są przedstawione w niniejszej pracy. Pozostałe, ze względu na charakter pracy, nie zostaną przedstawione.

Polecenie złożone (*compound statement*) — zawiera wiele poleceń zamkniętych w nawiasach { }.

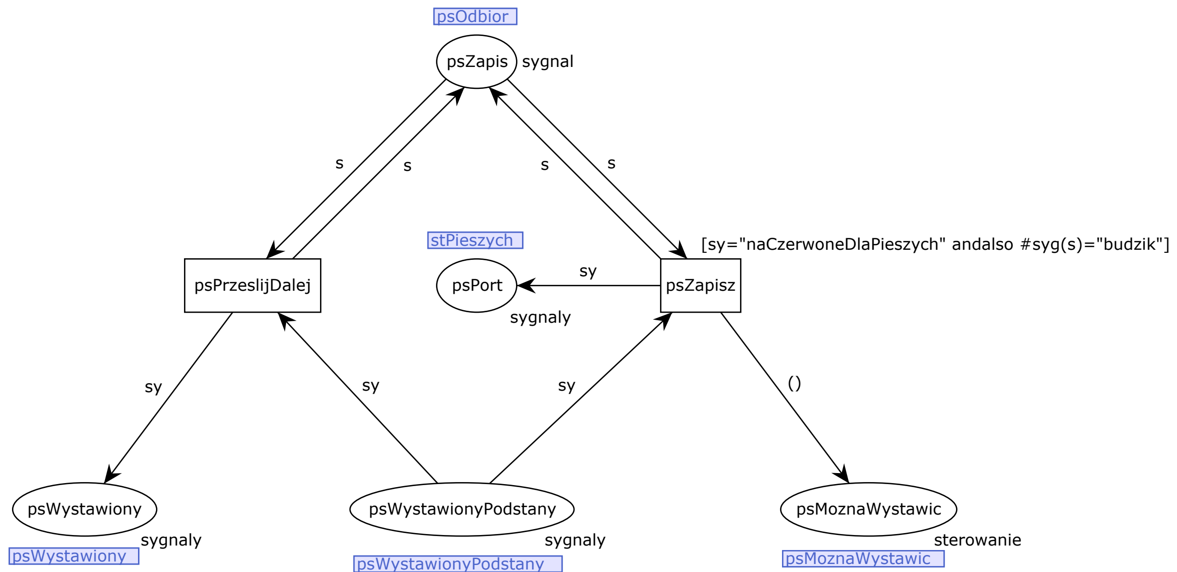
W zależności od zawartości może zostać przetłumaczone na poszczególne konstrukcje sieci Petriego z wykorzystaniem języka CPNML. Polecenie złożone określa również zakres widoczności zmiennych lokalnych w poleceniu złożonym. W sieci Petriego nie ma możliwości zdefiniowania zmiennych lokalnych w takim znaczeniu, w jakim występują one w UML. Pozostaje jedynie globalne zdefiniowanie i lokalne użycie. Jeżeli wykorzystuje się przesłanianie nazw zmiennych to należ zmodyfikować tak kod aby nazwy były unikalne.

Zapis (*save*) — zachowuje sygnały, żeby nie zostały odrzucone. Konstrukcja jest wykorzystywana gdy sygnały mogą przybywać w innej niż wymagana kolejności. Sygnał jest umieszczany na końcu kolejki, więc może zostać odrzucony gdy z niej wyjdzie (jeżeli nie zostanie odebrany albo ponownie zapisany).

Sieć Petriego dla tej konstrukcji składa się z przejścia (patrz rysunek 5.29). Pobiera ono sygnał (znacznik) z miejsca oczekiwania na odbiór (opis w podpunkcie dotyczącym wejścia) i wstawia go do miejsca początkowego konstrukcji związanej z kolejką (jako nadchodzący sygnał). Powoduje to dołączenie sygnału do końca kolejki. Wspomniane przejście jest wyzwalane przez sygnał oraz



Rysunek 5.28: Symbol zapisu



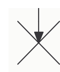
Rysunek 5.29: Reprezentacja zapisu w sieci Petriego

znacznik określający jaki sygnał ma być zapisany. Znacznik ten jest pobierany z maszyny stanowej (tak jak dla wejścia). Przejście wstawia znacznik do miejsca zezwalającego na udostępnienie kolejnego sygnału.

Koniec (*stop*) — kończy wykonywanie obecnej instancji. Usunięcie instancji klasy aktywnej jest możliwe jedynie z maszyny stanowej tej klasy, przez wykonanie działania kończącego. Jest ono wykonywane w następujący sposób:

– jeżeli instancja jest prostą maszyną stanową bez jakichkolwiek części, zostaje ona natychmiast zatrzymana

– jeżeli instancja zawiera części każda z instancji części zostanie potraktowana tak jak w powyższym punkcie. Również rozpatrywana instancja zostanie zatrzymana.



Rysunek 5.30: Symbol końca

W sieci Petriego konstrukcję tę można zamodelować przy pomocy przejścia. Powoduje ono zabranie znacznika sterowania z sieci a przez to jej dezaktywację. Przejście to blokuje również do-

stęp (wejście) do kolejki komunikatów (opis w podpunkcie poświęconym wejściu). Jeżeli maszyna posiada części to przejście wysyła do nich znaczniki wywołania destruktorów (albo sygnały zakończenia).

Powrót (*return*) — kończy wykonywanie metod lub podstanów i zwraca sterowanie do kontekstu wywołującego. W maszynie podstanowej powrót jest nazywany wyjściowym punkt łączącym (*exit connection point*) ponieważ umożliwia opuszczenie stanu złożonego.

Element ten w sieci Petriego jest realizowany w zależności od miejsca użycia. Jeżeli jest to metoda, to powrót jest przejściem, które zwraca sterowanie do miejsca w którym znacznik obiektu oczekuje na wywołania. Wysyła ono także odpowiedź do instancji, która wywołała metodę. Wynik jest zapisywany w znaczniku na łuku wyjściowym.

Wyjściowy punkt łączący w sieci Petriego jest reprezentowany przez miejsce. Ze względu na możliwość późniejszego sprawdzenia nazwy punktu wyjściowego jest ona zapisywana (w znaczniku) na łuku wyjściowym przejścia (zamiast nazwy poprzedniego stanu). Miejsce wyjściowe jest połączone fuzją z miejscem nadrzędnym. Jeżeli powrót nie ma nazwy to należy mu ją nadać.

Złącze (*junction*) — pozwala na łączenie części przejścia gdy ma ono wiele symboli między kolejnymi stanami (mechanizm typu *goto*). Złącze zawiera etykietę, która identyfikuje jego elementy (wejście oraz wyjście). Symbol złącza może mieć więcej niż jedną przychodzącą linię przepływu. Może znajdować się na różnych stronach diagramu.

W sieci Petriego konstrukcję tę można przedstawić w postaci miejsc połączonych fuzją.

Przepływ (*flow*) — łączy 2 elementy w przejściu.

W sieci Petriego reprezentowany przez łuki lub inne elementy związane z dwudzielnością grafu (analogicznie do krawędzi aktywności).

Proste przejście (*simple transition*) — symbolizuje przejście w widoku zorientowanym na stany. Konstrukcja ta zawiera informacje o zdarzeniu wyzwalającym (wejściu) dozorze oraz zadaniu. Można ją przetłumaczyć na sieć Petriego w sposób analogiczny do przedstawionego dla widoku zorientowanego na przejścia.

Wyrażenia (*expressions*) — podobne do wyrażeń w większości języków programowania. Mogą zawierać zmienne, literały, stałe oraz wywołania metod. Oprócz tradycyjnych wyrażeń następujące działania są również wyrażeniami to znaczy zwracają wartość:

– przypisanie (*assignment*) — przypisuje zmiennej wartość. Jeżeli jest to możliwe i ma sens przypisanie w sieci Petriego wykonywane jest na łuku wyjściowym przez wstawienie odpowiedniej wartości do znacznika. Aby było spójne należy pobrać wartości pól znacznika w łuku wejściowym przejścia. Wynika to z wykorzystania konstrukcji zapisywania wartości do znacznika. Wymaga ona ustawienia wszystkich pól.

Jeżeli tworzony jest nowy obiekt to trzeba wywołać konstruktor — wysłać znacznik do odpowiedniej klasy (strony). Otrzymana wartość (numer obiektu) jest zapisywana w polu reprezentującym zmienną. Operacja przypisania w sieci Petriego jest zazwyczaj poprawna jeżeli sieć jest poprawna. Błąd może wynikać z próby przypisania wartości spoza zakresu. Jest on wykrywany w trakcie analizy sieci.

– nowy (`new`) — polecenie to jest używane do tworzenia instancji klas aktywnych oraz pasywnych. Aby utworzyć instancję tej samej klasy co obecna można użyć słowa kluczowego `this`. Konstrukcja `new` zwraca odnośnik do utworzonego obiektu. Jeżeli występuje ograniczenie ilości instancji to powołanie nowego obiektu jest możliwe tylko w przypadku gdy ich liczba nie przekroczy wartości maksymalnej. W przeciwnym razie zostanie zwrócona wartość `NULL`.

  W sieci Petriego polecenie to jest interpretowane jako wysłanie przez przejście znacznika do konstruktora. Konstruktor zwraca identyfikator — numer obiektu w instancji strony. Jeżeli obiekt nie może zostać utworzony ze względu na ograniczenie liczby instancji, zwracana jest wartość `1` (szczegółowy opis konstrukcji w punkcie 6.1 — Modelowanie klas).

– wyrażenie warunkowe: `wynik=wyrażenie1? wyrażenie2 : wyrażenie3` gdzie pierwsze wyrażenie jest typu logicznego a drugie i trzecie są jednakowego typu. `wyrażenie1` jest wykonywane jako pierwsze. Jeżeli jest prawdziwe wykonywane jest `wyrażenie2` i udostępniany jest jego wynik jako rezultat wyrażenia warunkowego, w przeciwnym przypadku wykonywane jest `wyrażenie3` a jego wynik jest wartością wyrażenia warunkowego. wynik jest zmienną której przypisuje się wartość wyrażenia.

  Wyrażenie warunkowe można w sieci Petriego przedstawić analogicznie do rozstrzygnięcia. Można też umieścić odpowiednie wyrażenie na łuku wyjściowym przejścia. Ponieważ w sieci Petriego wyrażenie warunkowe nie zwraca wartości należy odpowiednio zmodyfikować wyrażenia wynikowe. Polega to na dodaniu instrukcji przypisania do wyrażenia.

  Konstrukcja taka wygląda następująco: `if wyrazenie1 then wynik=wyrazenie2 else wynik=wyrazenie3`

  Występuje kilka wyrażeń specjalnego dostępu do zmiennej lub tworzenia złożonych wartości.

– wyrażenie pola — używane w celu dostępu do pola struktur danych, czyli atrybutów obiektów: obiekt.atrybut.

  W sieci Petriego atrybut jest polem znacznika związanego z obiektem. Można się więc do niego odwołać przez nazwę: `#atrybut(zmienna)`, przy czym `zmienna` przechowuje znacznik obiektu (opis w punkcie 6.1 — Modelowanie klas). Konstrukcja ta zwraca wartość atrybutu. Jeżeli ma on być zapisany, to konstrukcja jest analogiczna do przypisania.

– wyrażenie indeksu — umożliwia dostęp do elementu indeksowanego typu danych, zazwyczaj tablicy albo łańcucha (na przykład: `l=tablica[i, j]`). Implementacja tablic w sieci Petriego jest problematyczna (opis w punkcie 3.2 — Predefiniowane typy danych). Dostęp do

konstrukcji z nią związanej zależy od jej reprezentacji. Jeżeli jest to lista to jej element można odczytać przy pomocy funkcji `nth(lista, numerElementu)`. Nie można (w taki przypadku) zmienić wartości elementu. Można natomiast dołączyć element na początek listy przy pomocy instrukcji `element::lista` lub na koniec `ins lista element`.

Konstrukcje dla innych rozwiązań są zastosowane przy konwersji elementów UML (np.: tablica obiektów), więc nie będą tutaj omawiane.

– wyrażenie struktury — używane do całościowego inicjalizowania złożonych struktur danych. Umożliwia ono przypisanie wartości w jednym działaniu. Nie trzeba, więc inicjalizować każdego pola osobno `struktura=(.wartość1, wartość2.)`.

W sieci Petriego konstrukcja polega na ustawieniu poszczególnych pól znacznika na łuku wyjściowym przejścia `{nazwa1=wartosc1, nazwa2=wartosc2}` w przypadku rekordu. Inicjalizacja innych typów złożonych nie zostanie tu przedstawiona ze względu na brak bezpośredniego wykorzystania w algorytmie.

– Wyrażenie **`this`** opisane w podpunkcie dotyczącym wyjścia.

Występuje też grupa wyrażeń, które podobnie jak dostęp do zmiennej, zależą od rzeczywistego stanu systemu. Są one również określane jako wyrażenia nakazujące (*imperative*):

– Wyrażenie **`any`** wstawia dowolną wartość danego typu `zmienna=any(typDanych)`.

W sieci Petriego można nie inicjalizować zmiennej. Zostanie jej wówczas przypisana dowolna wartość.

– Wyrażenie **`now`** zwraca obecną wartość czasu `czas=now`. Zmienna czas jest typu **`Time`**.

W sieci Petriego występuje funkcja **`time()`**, która zwraca bieżącą wartość zegara jako element typu **`time()`**.

– Wyrażenie *Pid* opisane w podpunkcie dotyczącym typ *Pid*.

– Wyrażenie state zwraca nazwę ostatnio odwiedzonego stanu w postaci łańcucha znaków. Jeżeli nie odwiedzono żadnego stanu zwracany jest pusty łańcuch.

`stan=state`

Jeżeli jest to wymagane w sieci Petriego można zamodelować opisywany mechanizm przy pomocy odpowiedniego pola w znaczniku zawierającego nazwę ostatnio odwiedzonego stanu (tak, jak w maszynie podstanowej). Wartość ustawiana jest przez łuk wyjściowy przejścia znajdującego się za miejscem reprezentującym stan. Należy zdefiniować dodatkową wartość związaną z brakiem stanu poprzedniego. Wartość ta jest ustawiana przez przejście początkowe maszyny stanowej.

– wyrażenie aktywnego budzika — używane do sprawdzenia czy dany budzik jest aktywny. Zwracana jest wartość logiczna. Budzik jest aktywny jeżeli jeszcze odlicza czas albo gdy skończył odliczanie, ale sygnał nie został jeszcze obsłużony (albo odrzucony).

`aktywny=active(budzik)`

W sieci Petriego można tę konstrukcję zamodelować przy pomocy dwóch przejść. Pierwsze może zostać odpalone, gdy występuje znacznik w konstrukcji budzika (miejsce „budzik"). Drugie z przejść może zostać odpalone, gdy brak jest znacznika w konstrukcji budzika.

Inne dostępne wyrażenia to:

– wyrażenie sprawdzenia zakresu — używane w celu sprawdzenia czy wartość spełnia warunki zakresu. Ma ono formę:

  `wartość in type Integer constants (0..9);`

  Wyrażenie sprawdzania zakresu zwraca wartość logiczną zależną od tego czy wartość zawiera się w danym typie.

  Wyrażenie to w sieci Petriego modeluje się przy pomocy konstrukcji: `in'cyfra(wartosc)`, przy czym `cyfra` jest kolorem zdefiniowanym dla danej sieci. Funkcja **`in`** musi zostać zadeklarowana w definicji typu z którym będzie używana:

  `textbfcolset cyfra=int with 0..9 declare in;`

– Wyrażenie kodu docelowego jest zależne od wybranego języka implementacji i zawiera kod języka implementacji, który nie jest analizowany przez parsera UML, lecz dodawany bezpośrednio do wygenerowanego kodu. Kod docelowy ma format: `[[szczegółyKoduDocelowego]]`. Kod docelowy może zawierać dowolne wyrażenie w języku implementacji, określonym przez kontekst UML.

  Aby przetłumaczyć takie wyrażenie na sieć Petriego należałoby opracować reguły jej tworzenia dla danego języka implementacji. Zagadnienie to wykracza poza zakres niniejszej pracy.

Każda klasa aktywna ma dostęp do 4 różnych wyrażeń *Pid* które mogą być postrzegane jako pośrednie atrybuty należące do instancji. Można je interpretować jako odnośniki do obiektów aktywnych — procesów.

W sieci Petriego są one zapisywane tak jak inne atrybuty klasy aktywnej (opis w punkcie 6.1 — Modelowanie klas). Ich format jest analogiczny do formatu adresu (opis w podpunkcie dotyczącym wejścia).

Poniżej zostaną opisane poszczególne argumenty, ich znaczenie oraz konstrukcje sieci związane z ustawianiem wartości. Odczyt wartości (w sieci) jest opisany w punkcie 6.1 — Modelowanie klas.

– Wyrażenie **`self`** zwraca wartość *Pid*, która odnosi się do danej instancji.

  W sieci Petriego wartość ta nadawana jest przez konstruktora.

– Wyrażenie **`sender`** zwraca wartość *Pid* odnoszącą się do obiektu, od którego dana instancja otrzymała ostatnio sygnał. Jeżeli żaden sygnał nie został otrzymany zwracana jest wartość **`NULL`**.

  W sieci Petrigo wartość jest przypisywana przez każdą konstrukcję odbierającą sygnał. Konstruktor ustawia wartość `{nazwaInstancji="~", numerObiektu= ~1}`.

– Wyrażenie **`parent`** zwraca wartość *Pid* odnoszącą się do obiektu, który utworzył daną instancję (rodzica). Jeżeli instancja została utworzona statycznie (przy starcie systemu) zwracana jest wartość **`NULL`**.

  W sieci Petriego odpowiednia wartość nadawana jest przez konstruktora.

– Wyrażenie **`offspring`** zwraca wartość *Pid*, która odnosi się do ostatnio utworzonego obiektu. Jeżeli nie został stworzony obiekt (przez daną instancję) zwracana jest wartość **`NULL`**.

  W sieci Petriego wartość nadawana jest przez konstrukcję tworzącą nowy obiekt. Konstruktor ustawia wartość `{nazwaInstancji="˜", numerObiektu= 1}`

W UML można zadeklarować zmienne typu *Pid*. Mogą one zawierać odnośniki do wszystkich rodzajów obiektów aktywnych, co może okazać się zbyt ogólne. Zmienne te można jednak zawęzić przez określenie ich typu jako interfejs albo klasa aktywna. Umożliwia to zweryfikowanie (przy statycznym sprawdzaniu) typu: czy wartość będzie odnosić się do instancji określonego rodzaju — na przykład przy próbie przypisania.

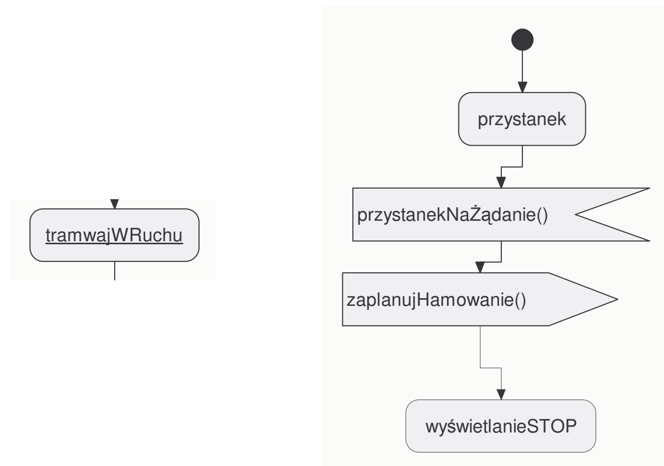
W sieci Petriego reprezentacja typu *Pid* jest taka sama, jak adresu. Jeżeli więc ma być sprawdzany typ wskazywanego obiektu należy go zapisać w osobnym polu znacznika — na przykład w postaci wartości typu wyliczeniowego.

Budzik — definiowany przez specjalny symbol lub odpowiednią składnię tekstową. W jego deklaracji możliwe jest określenie domyślnego interwału, czyli czasu pomiędzy ustawieniem budzika a jego zadziałaniem (tak jak w diagramie klas). Jeżeli odmierzanie ma zostać zakończone wykorzystuje się działanie wyłączające budzik. System monitoruje wszystkie aktywne budziki. O czasie zadziałania wysyłany jest sygnał budzika do procesu, który ustawił budzik. Proces ten będąc w odpowiednim stanie powinien odebrać sygnał budzika przez wejście. Tak jak zwykłe sygnały, sygnały budzika, które nie mogą być odebrane zostaną utracone.

W sieci Petriego konstrukcja budzika jest analogiczna do przedstawionej w podpunkcie dotyczącym wyrażenia sprawdzającego aktywność budzika. Należy jedynie zwrócić uwagę, że sygnał przeterminowania jest wysyłany na wejście kolejki sygnałów. Wiąże się to z umieszczeniem odpowiedniego wyrażenia na łuku wyjściowym przejścia generującego przeterminowanie. Musi ono tworzyć znacznik zgodny z formatem sygnału (opis w punkcie 6.1 — Modelowanie klas).

Stan złożony (*composite state*) — stan, który jest złożony z innych stanów oraz przejść. Podczas przebywania w jakimkolwiek podstanie (nie musi to być wyjściowy punkt łączący) stanu złożonego wyzwolenie przejścia stanu złożonego spowoduje wyjście ze stanu złożonego (oraz podstanów) do nowego stanu. Stan złożony może zawierać wiele etykietowanych punktów wejściowych oraz wyjściowych. Przejścia w podmaszynie mają wyższy priorytet niż przejścia w maszynie zewnętrznej. Odnosi się to zarówno do przejść wyzwalanych sygnałem jak i przejść wyzwalanych budzikiem.

W sieci Petriego maszyna podstanowa jest modyfikacją maszyny stanowej. Ze względu na możliwość wyjścia z niemal dowolnego stanu miejsca w pojedynczym przepływie (związanym z symbolem początkowym) są połączone fuzją. Dotyczy to jedynie miejsc z których sterowania może być



Rysunek 5.31: Symbol stanu złożonego oraz jego maszyna podstanowa

zabrane przez przejście nadrzędne. Sprowadza się to do miejsc reprezentujących podstany, w których maszyna podstanowa oczekuje na sygnał. Aby zapewnić odpowiedni przepływ sterowania znacznik zawiera dodatkowe pole, w którym zapisana jest nazwa ostatnio odwiedzonego stanu. Pole to jest typu wyliczeniowego, który zawiera nazwy wszystkich stanów danej maszyny. Wartość nadawana jest na łuku wyjściowym przejścia. Kolejne może zostać wyzwolone, jeżeli jest dla niego spełniony warunek nazwy (z uwzględnieniem innych kryteriów). Jest on sprawdzany na łuku wejściowym przejścia (przez wymuszenie konkretnej wartości w polu znacznika). Można, odpowiednio ustawiając warunki, zapewnić przepływ — (stan poprzedni)→(stan następny). Mechanizm opuszczenia stanu złożonego w (niemal) dowolnym momencie związany jest z przejściem, które ma początek w stanie nadrzędnym. Zabiera ono znacznik przepływu maszyny podstanowej.

Ze względu na fakt, że przejścia maszyny podstanowej mają wyższy priorytet, niż przejścia maszyny nadrzędnej trzeba zmodyfikować konstrukcję kolejki sygnałów. Sygnał udostępniany jest najpierw przejściom maszyny podstanowej przez wstawienie znacznika wystawianego sygnału do miejsca. Jest ono połączone fuzją z konstrukcjami związanymi z wejściem sygnału w podstanach.

Jeżeli sygnał nie zostanie pobrany zadziała konstrukcja odrzucania. Umieści ona sygnał w miejscu udostępnionym dla konstrukcji wejścia maszyny nadrzędnej. Jeżeli żadne wejście nie pobierze sygnału zostanie on zabrany przez przejście, które ma na łuku wejściowym warunek odrzucenia (`true` dla odbiorców z maszyny podstanowej). Znacznik jest umieszczany w miejscu, z którego mogą go zabrać wejścia maszyny nadrzędnej. Pozostała część konstrukcji kolejki sygnałów nie ulega zmianie.

Specjalizacja maszyny stanowej — maszynę stanową można specjalizować, albo bezpośrednio przez dziedziczenie pomiędzy maszynami stanowymi, albo przez specjalizowanie klasy aktywnej, która posiada daną maszynę stanową. Specjalizowana maszyna stanowa może posiadać dodatkowe cechy lub zmienione cechy oryginalnej maszyny stanowej. Dodatkowymi cechami mogą być: stany, przejścia, zmienne oraz inne encje, które mogą być zadeklarowane w maszynie stanowej. Aby umożliwić zmianę cechy przez specjalizowanie, musi ona być zadeklarowana jako wirtualna w oryginal-

nej maszynie stanowej. Wirtualna definicja może być przedefiniowana w specjalizowanej maszynie stanowej. W maszynie stanowej wirtualne (i tym samym możliwe do przedefiniowania) mogą być: przejścia, początek, wejście, dozór, zapis; metoda. Algorytm konwersji na sieć Petriego jest analogiczny do przedstawionego w punkcie 3.3 — Ogólne konstrukcje języka.

Ciało operacji — opis metody bez stanów. Jest to zazwyczaj działanie złożone z listy innych działań umieszczonych na diagramie tekstowym. Ciało operacji może mieć formę informacyjną, co oznacza, że określenie jak je wykonać nie jest wyrażone formalnie w języku UML, lecz w innym języku. W takim przypadku ciało metody będzie zawierać wyrażenie informacyjne zawierające nieformalny opis.

Można tę konstrukcję przedstawić w sieci Petriego (w zależności od zawartości) zgodnie z zasadami tłumaczenia zastosowanych działań. Sieć wynikowa znajduje się na osobnej stronie podstawionej pod przejście związane z wywołaniem metody.

Implementacja maszyny stanowej — funkcja zawierająca stany oraz pozostałe elementy niezbędne do zrealizowania sygnatury maszyny stanowej.

W sieci Petriego jest to strona zawierająca sieć reprezentującą maszynę stanową. Znajdują się w niej połączone ze sobą konstrukcje (uzyskane w wyniku konwersji) związane z modelowaniem działania maszyny stanowej. Odnosi się to między innymi do kolejki sygnałów opisanej w podpunkcie dotyczącym wejścia.

Zawartość (*internals*) — umożliwia podzielenie definicji klasy na część zorientowaną na sygnatury (deklaracje) oraz część zorientowaną na implementacje (definicje). Pozwala to na przechowywanie sygnatury klasy w innym pliku niż implementacja klasy. Ma to na celu umożliwienie oddzielnej obsługi wersji oraz dystrybucji dla sygnatur oraz implementacji w modelowaniu bazującym na komponentach.

W sieci Petriego deklaracje są powiązane z definicjami (z wyjątkiem predefiniowanych funkcji), wobec czego powyższa konstrukcja nie jest realizowana. Jest to jednak mechanizm związany bardziej z kompilacją niż weryfikacją, więc nie jest niezbędny do zamodelowania.

6. Algorytm translacji modelu szczegółowej specyfikacji systemu

Niniejszy rozdział opisuje diagramy pozwalające na modelowanie statycznego widoku budowanego systemu. Przedstawione tu konstrukcje umożliwiają określenie typów danych, powiązań pomiędzy fragmentami opisującymi zachowanie oraz projektowanie podziału funkcjonalności na spójne fragmenty.

6.1. Modelowanie klas

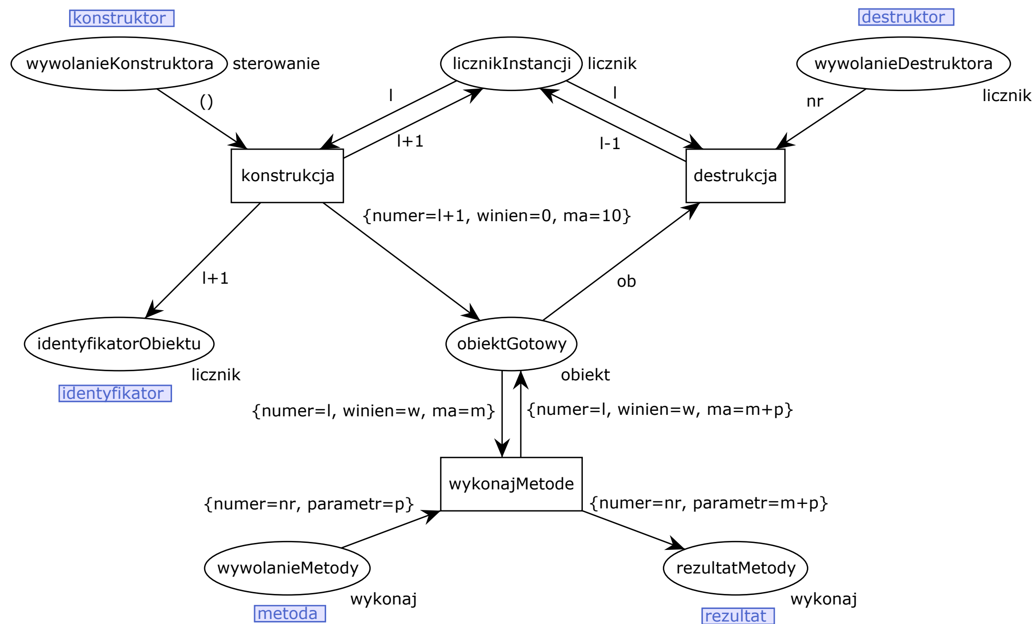
Jednym z ważniejszych etapów projektowania oprogramowania obiektowego jest opisanie rodzajów obiektów, które są częściami systemu.

Diagram klas (*class diagram*) opisuje statyczną 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, jednak ich forma może znacznie odbiegać od prostej analogii do diagramu klas.

Ogólnie strona przedstawiająca klasę zawiera sieci związane z realizacją konstruktora metod oraz destruktora (patrz rysunek 6.1). Sieci te mogą być (dla poprawienia czytelności) zrealizowane na osobnych stronach podstawionych pod przejścia.

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 maszyną stanową (klasa aktywna). Na stronie reprezentującej klasą znajduje się również licznik instancji. Jest on reprezentowany przez miejsce zawierające znacznik 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 zmniejsza natomiast wartość licznika instancji.

Pierwszym krokiem konwersji jest utworzenie strony, na której mogą znaleźć się następujące konstrukcje z UML:



Rysunek 6.1: Sieć Petriego reprezentująca klasę

Klasa (*class*) — abstrakcja grupy obiektów, które mają takie same właściwości (atrybuty), zachowania (metody) strukturę i relacje (patrz rysunek 6.2). Klasę można powołać do życia tworząc obiekty (jeżeli nie jest to klasa abstrakcyjna), które mają takie same właściwości. Ogólnie klasa nie posiada swojego własnego sterowania (jest pasywna w odróżnieniu od klasy aktywnej), które otrzymuje w celu wykonania operacji.

Rysunek 6.2: Symbol klasy

W sieci Petriego definicję klasy przedstawia się na osobnej stronie z którš elementy korzystające z obiektów łączą się przy pomocy fuzji miejsc (patrz rysunek 6.1).

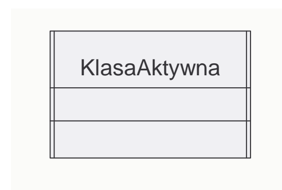
Klasa abstrakcyjna (*abstract class*) jest klasą, której instancji nie można bezpośrednio powołać do życia (patrz rysunek 6.3). Jest ona elementem teoretycznym, którego właściwości zostały opisane w punkcie 3.3 — Ogólne konstrukcje języka. Zastosowanie klasy abstrakcyjnej związane jest z użyciem mechanizmów obiektowych - dziedziczenia.

Podczas tworzenia sieci dla klasy abstrakcyjnej należy postępować zgodnie z zasadami konwersji dla elementu teoretycznego. Klasa może posiadać parametry kontekstowe, które są precyzowane przez dziedziczącą ją klasę.

Rysunek 6.3: Symbol klasy abstrakcyjnej

Wirtualność określa czy klasa może zostać przedefiniowana. Ma to zastosowanie tylko wtedy gdy klasa zawiera inną klasę. Sieć Petriego w takim przypadku jest tworzona według algorytmu opisanego w punkcie 3.3 — Ogólne konstrukcje języka.

Klasa aktywna (*active class*) jest klasą posiadającą zachowanie określone w maszynie stanowej (patrz rysunek 6.4). Jest to fundamentalna konstrukcja modelowania zachowań czasu rzeczywistego w UML. Jeżeli klasa aktywna posiada wiele części, każda z nich wykonuje się asynchronicznie i współbieżnie z innymi częściami w systemie. Taka semantyka umożliwia umieszczenie modelu w fizycznie rozproszonym środowisku przez co nie będzie on zależny od wykonania na pojedynczym procesorze z współdzielonym dostepem do pamięci.



Rysunek 6.4: Symbol klasy aktywnej

Sieć Petriego dla tej konstrukcji jest siecią maszyny stanowej (opis w punkcie 5.2 — Modelowania zachowania) zmodyfikowaną o elementy charakterystyczne dla klasy — atrybuty (w znaczniku), konstruktor (zamiast początku), licznik instancji itp. opisane przy omawianiu klasy pasywnej.

Struktura klasy aktywnej jest definiowana na jednym lub wielu diagramach struktury złożonej opisanych w punkcie 6.2 — Modelowanie architektury.

Klasa zewnętrzna (*external*) jest klasą, której szczegóły implementacyjne nie są przedstawiane w modelu. W związku z powyższym nie jest ona tłumaczona na sieć Petriego — wymaga zdefiniowania w oparciu o informacje z określonego w innym miejscu modelu.

Każda klasa może posiadać atrybuty oraz metody.

Atrybut (*attribute*) jest cechą która może przechowywać jeden lub wiele wartości podczas pracy systemu.

W sieci Petriego atrybuty proste są polami znacznika reprezentujacego 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 byż powiązany przez:

– zwykłe powiązanie,

– agregację — słowo kluczowe **shared**,

– kompozycję — słowo kluczowe **part**.

Opis sieci Petriego dla powyższych konstrukcji przedstawiono w punkcie 3.5 — Relacje w UML.

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.

Atrybut może mieć wartość domyślną:

`Ilość: Integer=4`

W sieci Petriego jest ona ustawiana przez fragment związany z konstruktorem.

Atrybuty mogą występować w liczebności większej niż 1 zapisywanej, w postaci: `nazwatextbf: typ[liczebność]`. Liczebność (podczas pracy systemu) może się zawierać w granicach określonych przez wyrażenie `liczebnośćMinimalna ..liczebnośćMaksymalna`. Symbol `*` 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`) nie stosuje się kontenera.

W zależności od dostępnych bibliotek typów danych, typ kontenera może być różny. Zazwyczaj poszczególne generatory kodu będą dostarczać różnych typów kontenerów dla zapewnienia odpowiedniej integracji z językiem docelowym. Jeżeli nie została dołączona dedykowana biblioteka użyty zostanie typ **String**.

W sieci Petriego konstrukcja związana z liczebnością zależy od użytego typu kontenera. Rozwiązanie można wybrać z przedstawionych w punkcie 3.2 — Predefiniowane typy danych.

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. W sieci reprezentującej kontener (klasa zawierająca) znajduje się licznik instancji (obiektów). Jego działanie polega na zliczaniu wywołań konstruktora oraz destruktora. Wywołanie uzależnione jest od bieżącej liczby instancji oraz jej ograniczeń. Rozwiązanie składa się z miejsca oraz trzech przejść (rysunek ref). W miejscu znajduje się znacznik, w którym zapisana jest liczba instancji. Jedno z przejść odpowiada za zliczanie wywołań konstruktora. Pobiera ono 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 powinien (ze względu na ograniczenia) być utworzony to inne przejście przesyła znacznik do miejsca wskazującego na niepowodzenie stworzenia kolejnej instancji. Trzecie z przejść odpowiada za wy-

wołanie destruktora. Dekrementuje ono wartość znacznika instancji i przesyła otrzymany znacznik (z numerem obiektu) do destruktora.

W UML można zdefiniować początkową liczbę instancji, które zostaną stworzone automatycznie, gdy zawierająca je encja będzie tworzona (później ilość ta może się zmieniać). Jeżeli nie jest podana liczebność początkowa, ilość początkowo tworzonych instancji będzie równa dolnej granicy liczebności części. Jeżeli nie podano liczebności, automatycznie zostanie utworzona jedna instancja a ilość równoczesnych instancji nie będzie ograniczona od góry.

```
tablica: Integer[*]/5
```

Liczebność początkowa ustawiana jest przez konstruktora klasy zawierającej. Wysyła on odpowiednią liczbę znaczników do fragmentu sieci związanego z kontenerem i zapisuje otrzymane identyfikatory w atrybucie.

Atrybut, który jest parametrem typu klasy zdarzenia (*Event Class*) może zostać zadeklarowany jako wirtualny (`virtual`) dla umożliwienia jego przedefiniowania przy specjalizowaniu sygnatury (sygnał, budzik, metoda).

Konstrukcja sieci Petriego w takim przypadku jest analogiczna do przedstawionej w punkcie ref — Ogólne konstrukcje języka.

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

```
/moduł: Real
```

Sieć Petriego dla takiego atrybutu konstruuje się zgodnie z algorytmem dla elementu wyprowadzanego.

Atrybut statyczny (*static*) — jest atrybutem który, jest związany bardziej z klasą niż instancją. Oznacza to, że istnieje tylko 1 instancja atrybutu współdzielona przez wszystkie instancje danej klasy.

```
static łańcuch: String
```

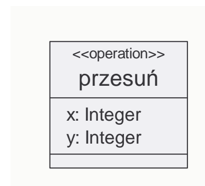
W sieci Petriego nie ma możliwości przechowywania wartości zmiennej globalnej. Problem ten można ominąć przez utworzenie miejsca połączonego fuzją między instancjami w którym będzie znajdował się znacznik przechowujący wartość atrybutu statycznego. Jeżeli, pewna instancja będzie potrzebować jego wartości, 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 konstrukcja jest analogiczna do deklaracji stałej — opis w punkcie 3.2 — Predefiniowane typy danych. Należy jednak zwrócić uwagę aby nie występowały stałe o tych samych nazwach gdyż jest to deklaracja globalna. Wprawdzie nie jest to dokładne odwzorowanie mechanizmu, ale wydaje się wystarczające do zastosowania w sieci Petriego. Dla pełnej zgodności należałoby zadeklarować atrybut stały tak jak statyczny.

Operacja (*operation*) jest deklaracją wskazującą, że instancje klasy będą w stanie przechwycić wywołania, które są zgodne z sygnaturą operacji (patrz rysunek 6.5). Może ona zostać zaimplementowana albo przy pomocy opisu tekstowego (*operation body*) albo przy pomocy maszyny stanowej. Metoda (implementacja operacji) jest wykonywana, jeżeli zostanie wywołana. To oznacza, że jeżeli odbiorca jest instancją pasywną implementacji, to 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.

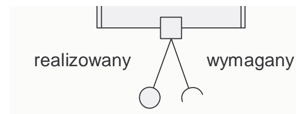


Rysunek 6.5: Symbol operacji

W sieci Petriego metody nie są deklarowane. Ich definicje znajdują się na stronie zawierającej definicję klasy (patrz rysunek 6.1).

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). 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. Żeby metoda mogła zwrócić wynik, musi posiadać adres nadawcy. Przejście rozpoczynające działanie metody (na stronie klasy) pobiera go i umieszcza w osobnym miejscu (zapisany w znaczniku). Jest on z niego pobierany przy zakończeniu metody (przejście reprezentujące powrót).

Port (*port*) jest nazwanym punktem komunikacji (*interaction point*) klasy aktywnej (patrz rysunek 6.6). Określa on zrealizowany (przez klasę) interfejs oraz wymagania dotyczące interfejsów innych klas. 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. Występują 2 rodzaje portów: zachowania (*behavior port*) oraz nie związany z zachowaniem (*non-behavior port*). Różnica pomiędzy tymi dwoma rodzajami wynika z faktu, że port zachowania jest bezpośrednio powiązany z maszyną stanową klasy natomiast port nie związany z zachowaniem wymaga połączenia przy pomocy łączników i przeważnie jedynie przekazuje komunikację z zewnątrz klasy do pewnych wewnętrznych części klasy. Wszystkie sygnały wysłane do portu zachowania są konsumowane przez zachowanie klasy. Możliwe jest wysłanie sygnału na port bez jawnego określenia jego odbiorcy. Sygnał taki zostanie odebrany przez instancję, która jest połączona do tego portu. Interfejs (zrealizowany/wymagany) portu może zawierać wskazania do interfejsów, ale również do listy sygnałów, sygnału, atrybutu lub metody.



Rysunek 6.6: Symbol portu wraz z interfejsami

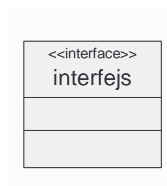
W sieci Petriego port jest reprezentowany przez miejsce pośredniczące w komunikacji, przez które przesyłane są sygnały. Jego kolor jest kolorem (typem) znacznika odpowiadającego sygnałowi. Jeżeli port ma przesyłać wiele różnych sygnałów, to w sieci Petriego należy zdefiniować jego kolor tak, jak dla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

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

Porty mogą zostać zadeklarowane jako prywatne lub publiczne dla rozróżnienia czy port jest udostępniony na zewnętrz, czy jest używany jedynie wewnętrznie.

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 (patrz rysunek 6.7).

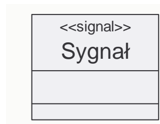


Rysunek 6.7: Symbol portu wraz z interfejsami

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

Interfejs może podlegać specjalizacji, oraz posiadać parametry kontekstowe. Interfejsy mogą być ze sobą powiązane, co umożliwia definiowanie protokołów lub kontraktów pomiędzy klasami, które realizują uczestniczące w nich interfejsy. Powiązanie interfejsów tworzy relację między nimi. Oznacza to, że jeżeli jeden z interfejsów zostanie wskazany (na przykład w porcie lub związany z łącznikiem), drugi z interfejsów zostanie automatycznie „podłączony” w przeciwną stronę (realizowany powiązany z wymaganym).

Sygnał (*signal*) jest jednym z podstawowych środków komunikacji w UML (patrz rysunek 6.8). Sygnał jest asynchronicznym komunikatem, który jest przesyłany pomiędzy klasami aktywnymi. Może on przekazywać dane, które muszą być zgodne z zadeklarowanym typem parametru sygnału.



Rysunek 6.8: Symbol sygnału

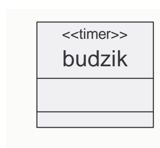
W sieci Petriego sygnał jest reprezentowany przez znacznik, którego pola są parametrami sygnału. Znacznik jest zadeklarowany jako rekord a typy jego pól odpowiadają typom parametrów sygnału. Jedno z pól przechowuje nazwę sygnału. Konstrukcja związana z adresowaniem sygnału przedstawiona jest w punkcie 5.2 — Modelowanie zachowania.

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 bardziej jako wybór.

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

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

Budzik (*timer*) — jest zdarzeniem, które w taki sam sposób jak sygnał może wyzwolić przejście (patrz rysunek 6.9). Budzik jest ustawiany przez maszynę stanową i po określonym czasie (*timeout*) sygnał budzika może zostać odebrany przez tą samą maszynę stanową. Gdy budzik deklarowany jest tekstowo, jest też możliwe, ustawienie domyślnej wartości czasu po którym wyśle sygnał. Pozwala to na aktywowanie budzika bez ustawiania czasu. Można również sparametryzować go (jak sygnał) co umożliwia wielokrotne wykorzystanie budzika tego samego typu bez kasowania aktywowanego wcześniej budzika. Oznacza to, że wiele budzików z różnymi wartościami parametru może być aktywnych w tym samym czasie.



Rysunek 6.9: Symbol budzika

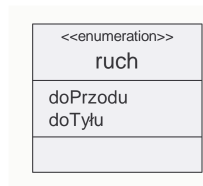
W sieci Petriego konstrukcją budzika można zdefiniować na osobnej stronie i podstawiać tak jak inne klasy. Należy wówczas przesłać wartość opóźnienia w znaczniku wyzwalającym budzik. Jest ona wykorzystywana na łuku wyjściowym ustawiającym opóźnienie (`@+wartosc`). Jeżeli wartość nie jest przesyłana (ustawiona jest na stałe) to konstrukcję można zinterpretować jako budzik z wartością domyślną.

Typy danych (*datatype*) — są używane w 2 celach: do opisu dostępnych typów podstawowych, oraz do opisu zdefiniowanych przez użytkownika typów wyliczeniowych. Użytkownik może zdefi-

niować swoje typy podstawowe w modelu, ale może to powodować problemy z generacją kodu. Typ wyliczeniowy definiuje zbiór wartości przez ich przedstawienie (wyliczenie). W każdym przypadku typ danych może również dodatkow zawierać zachowanie, które jest określone przez metody.

Jeżeli typ danych nie określa zachowania to można całą definicję zawrzeć w sieci Petriego, w formie deklaracji. Jeżeli typ posiada zachowanie to sieć jest analogiczna do tej dla konstrukcji klasy.

Typ wyliczeniowy (*enumerated*) jest typem w którym wartościami formalnymi są nazwy logiczne (patrz rysunek 6.10). Można do nich dodatkowo dołączyć wartości określone przez proste wyrażenie.



Rysunek 6.10: Symbol typu wyliczeniowego

Dostępne są domyślne operacje:

- (nie)równość (**==**, **!=**)
- relacji (**<, >, <=**, **>=**)
- przypisanie (**=**)

Możliwa jest konwersja pomiędzy typem całkowitym i wyliczeniowym. W tym celu używa się operacji rzutowania - **cast**.

W sieci Petriego typ wyliczeniowy deklaruje się przy pomocy konstrukcji:

```
colset ruch=with doPrzodu|doTylu;
```

Jeżeli ma być wykorzystywana operacja rzutowania to należy zdefiniować odpowiednie funkcje (po jednej na konwersję w każdą stronę). Jej definicja może wyglądać następująco:

```
fun cast1(w)=case w of doPrzodu=>10|doTylu=>  10;
```

Można ją wywołać na łuku wyjściowym przejścia. Jeżeli mają być wykorzystywane warunki nierówności to należy zwróciż uwagę, że występują różnice pomiędzy UML a CPN:

UML	CPN
==	**=**
!=	**<>**
=	przez wartość wyjściową funkcji

Jeżeli mają być wykorzystane operatory nierówności, to należy posłużyć się konstrukcją if/then/else.

Literał (*literal*) jest definicją elementu określonego typu. Literał jest własnością tego typu. Oprócz posiadania nazwy (co jest cechą wszystkich definicji), literał może mieć też związaną z nim

wartość, co umożliwia wykorzystanie go w wyrażeniach arytmetycznych. Definicja ta może zostać w sieci Petriego zinterpretowana analogicznie do tej we wcześniej przedstawionej konstrukcji.

Wybór (*choice*) — typ przechowujący jedną z wartości różnych typów. Wybór jest dokonywany, gdy przypisywana jest wartość zmiennej.

```
choice łańcuchAlboCałkowita
{
        String łańcuch;
        Integer całkowita;
}
```

W sieci Petriego typ ten można zamodelować przy pomocy deklaracji:

```
colset lancuchAlboCalkowita=union lancuch: typLancuch+calkowita:
typCalkowita;
```

gdzie `typLancuch` oraz `typCalkowita` są odpowiednio określonymi wcześniej kolorami.

Dla każdego potencjalnego pola typu mozna użyć operatora **`IsPresent()`**, który sprawdza obecność pola. W sieci Petriego nie ma odpowiednika tego operatora. Trzeba więc zadeklarować rekord, w którym pierwsze pole będzie unią natomiast drugie będzie zawierać informację o typie przechowywanej wartości. Można do tego celu wykorzystać typ wyliczeniowy (np. `r|c|b`).

Synonim (*syntype*) jest typem danych bazującym na innym typie danych — rodzicu. Opisywane typy nie są różne w sensie kompatybilności oraz literałów. Literały synonimu są jednakowe, lub podzbiorem literałów rodzica. Synonim może być postrzegany jako odsyłacz do innego typu; odsyłacz, który może być ograniczony (zawężony).

```
syntype cyfry=Integer constants (0..9);
syntype polowa=Integer constants (4, 5, 6);
syntype dodatnie=Integer constants (>0);
syntype alias=cyfry;
```

W sieci Petriego deklaracje mogą wyglądać następująco:

```
colset cyfry=int with 0..9;
```

lub

```
colset polowa=subset cyfry textbfwith [4, 5, 6];
```

Dla bardziej skomplikowanych przedziałów należy zdefiniować funkcję. Konstrukcja taka przedstawiona jest poniżej.

```
colset calkowity=int;
fun selekcja(n)=n>0;
colset dodatnie=subset calkowity by selekcja;
```

lub, jako alias `colset alias=cyfry;`

Elementy typu maszyna stanowa został opisany w punkcie 5.2 — Modelowanie zachowania.

Element typu stereotyp został opisany w punkcie 7.1 — Rozszerzalność.

Element typu pakiet został opisany w punkcie 3.6 — Modelowanie pakietów.

Element typu artefakt został opisany w punkcie 6.4 — Modelowanie rozmieszczenia.

Relacje powiązania, połączania, złożenia, zależności, uogólnienia, wykonania oraz wskazania są opisane w punkcie 3.5 — Relacje w UML.

Relacja rozszerzenia jest opisana w punkcie 4.1 — Modelowanie przypadków użycia.

6.2. Modelowanie architektury

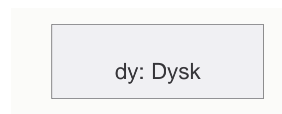
Podczas modelowania architektury opisywana jest wewnętrzna struktura klas z punktu widzenia komunikacji. Polega to na łączeniu atrybutów klasy (w tym kontekście odnoszących się do części) łącznikami oraz określaniu, które sygnały mogą być wysyłane tymi łącznikami. Struktura części i łączników jest nazywana architekturą klasy.

Diagram struktury złożonej (*internal structure diagram* (UML)/*Architecture Diagram* (TAU 2.1)/*Composite Structure Diagram* (TAU 2.4)) opisuje dynamiczną strukturę wewnętrzną klasy aktywnej związaną z innymi klasami aktywnymi. Są one reprezentowane jako części znajdujące się w innej klasie (kontenerze). Diagram struktury złożonej opisuje komunikację wewnątrz klasy aktywnej przez pokazanie łączników pomiędzy portami komunikacji oraz częściami.

W sieci Petriego wynik konwersji elementów tego diagramu wstawiany jest na stronie definiującej klasę przedstawioną na diagramie.

Na diagramie mogą występować następujące elementy:

Część (*part*) reprezentuje jedną lub więcej instancji będących w posiadaniu rozważanej klasy (patrz rysunek 6.11). Ponieważ część jest obiektem jej reprezentacja w sieci Petriego jest związana z konstrukcją przedstawioną w punkcie 6.1 — Modelowanie klas.



Rysunek 6.11: Symbol części

Jeżeli pominięto referencję do klasy to jest to związane z definicją części, z wplecioną (*inline*) definicją klasy. Określenie części w ten sposób oznacza, że definicja klasy nie jest oddzielna od jej użycia, co sprawia, że opis jest bardziej spójny, jednak z drugiej strony mniej nadaje się do ponownego użycia.

W sieci Petriego konstrukcję tę tłumaczy się tak jak inne konstrukcje obiektowe. Jest to związane z bezpośrednim wykorzystywaniem definicji „abstrakcyjnej" przez rzeczywisty obiekt w reprezentacji używanej, w sieci.

Łącznik (*connector*) określa medium, które umożliwia komunikację pomiędzy częściami klasy aktywnej (łączy porty) lub między otoczeniem klasy aktywnej a jedną z jej części. Łącznik może przesyłać określone informacje w jedną lub dwie strony (może mieć 2 zwroty). Informacja ta może być reprezentowana przez: sygnał, atrybut, listę sygnałów oraz interfejs.

Linia łącznika określa ścieżkę komunikacji pomiędzy dwoma punktami końcowymi, na przykład portami części, zachowania lub ramki diagramu.

Struktura klasy aktywnej może zawierać jawne lub niejawne linie łączników. Jawne łączniki są widoczne i można się do nich odwołać w przeciwieństwie do niejawnych.

Niejawne łączniki są wyznaczane na podstawie wszystkich pasujących zrealizowanych oraz wymaganych interfejsów wykorzystywanych w:

– portach części klasy zawierającej

– portach klasy zawierającej

– portach zachowań klasy zawierającej

Jeżeli port ma jawne łączniki to nie zostaną podłączone do niego łączniki niejawne.

Linia łącznika zawiera 2 pola tekstowe (łącznik w jedną stronę) albo 3 pola tekstowe (łącznik w obie strony).

Środkowe pole określa nazwę łącznika natomiast pole znajdujące się na końcu linii określa obszar listy sygnałów. Każdemu grotowi strzałki odpowiada jeden obszar listy sygnałów. Może on pozostawać pusty.

W sieci Petriego łącznik jest przejściem łączącym 2 miejsca (porty). Przejście łączy się łukami, których zwrot jest zgodny ze zwrotem łącznika. Etykietą łuku jest zmienna, umożliwiająca przesłanie określonych sygnałów. Jeżeli łącznik zadeklarowany jest jawnie, to typ zmiennej dla etykiety łuku jest typem (kolorem) będącym rekordem typów sygnałów zawartych w obszarze listy sygnałów. Jeżeli łącznik nie został jawnie określony, to buduje się go według przedstawionych wyżej zasad. Następnie tworzy się sieć tak, jak w przypadku jawnej deklaracji łącznika. Jedno przejście odpowiada za 1 zwrot, więc jeśli łącznik ma 2 zwroty potrzebne są 2 przejścia.

Port zachowania (*behavior port*) służy przede wszystkim do komunikacji pomiędzy zachowaniem a częścią w klasie aktywnej. Jest on portem maszyny stanowej (przedstawionej jako odnośnik) definiującej zachowanie klasy aktywnej. Port zachowania umożliwia określenie interfejsu komunikacji maszyny stanowej. Port zachowania odnosi się do pojedynczej maszyny stanowej opisywanej klasy. Na pojedynczym diagramie może znajdować się wiele portów zachowania, co oznacza, że odnoszą się one do tego samego zachowania.

W sieci Petriego port zachowania jest miejscem, które jest połączone z przejściem. Pod przejście podstawiona jest (odpowiednik odnośnika) strona zawierająca definicję maszyny stanowej. Typ miejsca definiuje się analogicznie do typu portu. Etykiety łuków definiuje się analogicznie do definicji łuków w konstrukcji łącznika.

W diagramie struktury złożonej używana jest relacja zależności pomiędzy częściami aby pokazać, że jakaś część zależy od innej. Przeważnie relacja ta wskazuje na zależność tworzenia pomiędzy częściami — instancje danej części mogą tworzyć instancje innej części. Opis sieci Petriego dla tej konstrukcji zamieszczony jest w punkcie 3.5 — Relacje w UML

6.3. Modelowanie komponentów

Modelowanie komponentów polega na określaniu kluczowych komponentów systemu i modelowaniu ich interfejsów oraz relacji. W tej perspektywie zwięźle opisuje się system przez ukrycie szczegółów w komponencie, i udostępnienie jedynie małego zbioru dobrze zdefiniowanych interfejsów. Zależności pomiędzy komponentami przedstawiane są również w zminimalizowany sposób (słabe powiązanie).

Diagram komponentów (*component diagram*) opisuje statyczną strukturę systemu przy pomocy zbioru komponentów oraz ich zrealizowanych i wymaganych interfejsach. Przedstawiane są też relacje między komponentami. Inne elementy modelu takie jak klasy czy artefakty mogą również występować na diagramie komponentów dla zobrazowania ich relacji z komponentami.

Na diagramie komponentów mogą się znaleźć:

– artefakt — opisany w punkcie 6.4 — Modelowanie rozmieszczenia

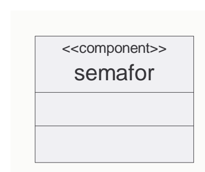
– klasa, interfejs (zrealizowany/wymagany), port — opisane w punkcie 6.1 — Modelowanie klas

– relacje: powiązania, połączenia, złożenia, zależności, uogólnienia, wykonania, wskazania — opisane w punkcie 3.5 — Relacje w UML.

– komponent — mała, zwięzła część systemu, która zapewnia określone usługi (patrz rysunek 6.12). Są one przedstawione przez zrealizowane interfejsy. Dostęp do komponentu możliwy jest tylko przez nie. Komponent może być zależny od innych usług, co jest zaznaczone przez wymagane interfejsy.

  Implementacja komponentu czyli jego zachowanie oraz architektura nie powinny być udostępnione klientom. Gdy udostępnione są jedynie interfejsy, komponent może zostać zastąpiony przez inny, zupełnie inaczej zaimplementowany, bez zmiany klienta.

  Różnica pomiędzy klasą a komponentem w UML jest minimalna — mogą one być używane wymiennie (w analogiczny sposób). Komponent jest podklasą klasy w metamodelu. Komponent i klasa mogą mieć atrybuty, metody, strukturę złożoną (przedstawioną na diagramie struktury złożonej), porty, interfejsy itp. Podstawową przesłanką istnienia komponentu jest zapewnienie terminologii i podkreślenie tych cech, które są najważniejsze w modelowaniu opartym na komponentach. Oznacza to możliwość pokazania jak komponent jest zrealizowany lecz również, określenia jego interfejsów zrealizowanego oraz wymaganego.



Rysunek 6.12: Symbol komponentu

W związku z podobieństwem komponentu do klasy można wykorzystać konstrukcję klasy (opisanej w punkcie 6.1 — Modelowanie klas) jako reprezentanta komponentu w sieci Petriego.

6.4. Modelowanie rozmieszczenia

Modelowanie rozmieszczenia opisuje architekturę pracy systemu. Przedstawia ona jak fragmenty oprogramowania (artefakty) są rozmieszczane w węzłach reprezentujących fizyczne zasoby obliczeniowe. Model rozmieszczenia opisywany jest na diagramie rozmieszczenia. Przedstawia on zbiór rozmieszczonych artefaktów w zbiorze połączonych węzłów.

W sieci Petriego jako diagram rozmieszczenia można traktować stronę pokazującą hierarchię.

Na diagramie rozmieszczenia mogą się znaleźć:

– artefakt (*artifact*) — fizyczny fragment informacji, która jest używana albo wytwarzana w procesie rozwijania oprogramowania. Przykładami artefaktów są pliki źródłowe, skrypty, biblioteki oraz wykonywalne programy.

  Artefakty są elementami podobnymi do klas — mogą mieć atrybuty oraz metody. Artefakty mogą również występować w następujących relacjach: zależności (dla elementu), uogólnienia (pomiędzy artefaktami), złożenia (zazwyczaj wobec innych artefaktów). Artefakt jest również przestrzenią nazwy i może przez to posiadać inne elementy modelu.

  W sieci Petriego artefakt jest reprezentowany przez stronę zawierającą definicje związane z elementami artefaktu.

– węzeł (*node*) — nazwany zasób obliczeniowy (zazwyczaj konkretny komputer). Węzły mogą być powiązane w celu modelowania topologii sieci.

  Ze względu na charakter węzła (element środowiska) jego modelowanie przy pomocy sieci Petriego jest problematyczne (interpretacja wyników analizy formalnej).

– środowisko wykonania (*execution environment*) — szczególny rodzaj węzła dostarczający środowisko wykonania umieszczonym w nim artefaktom. Składa się ono zazwyczaj ze zbioru usług wymaganych przez artefakt podczas pracy. Przykładem może być serwer J2EE przygotowany do rozmieszczenia ziaren (*beans*) J2EE.

  W sieci Petriego jest to odpowiednio przetłumaczony model środowiska.

  *s*pecyfikacja rozmieszczenia (*deployment specification*) — używana do określenia zbioru początkowych właściwości ustalających parametry wykonywania artefaktu. Specyfikacja rozmieszczenia powiązana jest z artefaktem przez relację zależności.

  W sieci Petriego konstrukcję tę można interpretować jako odpowiednie ustawienie znakowania początkowego reprezentującego rozważane opcje.

– klasa — opisana w punkcie 6.1 — Modelowanie klas.

– relacje:

  - rozmieszczenia (*deployment*) — szczególny rodzaj zależności używany do umieszczenia artefaktu. Będzie się on wykonywał w kontekście węzła (gdzie został umieszczony).

    W sieci Petriego relacja ta jest modelowana przez podstawione przejście. Jest ono łączone (wejście oraz wyjście) z miejscem reprezentującym zasoby obliczeniowe. Jego podstroną jest strona zawierająca definicję artefaktu.

  - wskazania, powiązania, połączenia, złożenia, uogólnienia, zależności opisane w punkcie 3.5 — Relacje w UML.

7. Zaawansowane konstrukcje UML

UML zapewanijšc możliwość wspomogania projektowania oprogramowania dostarcza również mechanizmy, które mogą rozszerzać jego funkcjonalność. W rozdziale zostaną opisane rozwiązania, które pozwalają na definiowanie takich modyfikacji.

7.1. Rozszerzalność

UML jest językiem, który może być w pewnym zakresie modyfikowany. Występują w nim predefiniowane mechanizmy rozszerzania konstrukcji oraz specjalizacji dla konkretnych zastosowań.

Rozszerzalność definiuje następujące pojęcia:

Metaklasa — używana do kategoryzowania zbioru elementów przechowywanych w repozytorium UML. Definiowana w metamodelach przy użyciu symbolu klasy — nazwa klasy jest poprzedzona określeniem «`metaclass`».

Metamodel — szczególny rodzaj pakietu klas modelu UML, który jest używany do opisu informacji przechowywanych w repozytorium narzędzia. Pakiet jest metamodelem, jeżeli jego nazwę poprzedza słowo kluczowe «`metamodel`».

Metamodel jest zbiorem metaklas, metaatrybutów itp. które określają pojęciowy widok informacji (definicje UML) przechowywanych w repozytorium modelu. Metamodel może zostać użyty jako podstawa do definicji profilu.

Zestaw narzędzi UML umożliwia reprezentowanie różnych metamodeli każdy dający inny widok (rodzaj diagramu) określonego modelu.

Stereotyp — służy do poszerzania zbioru informacji, które mogą być przechowywane w modelu dla danej encji. Dodatkowa informacja jest opisywana przez atrybuty stereotypu. Nazwa stereotypu poprzedzona jest słowem kluczowym «`stereotype`».

Profil — szczególny rodzaj pakietu, identyfikowanym przez słowo kluczowe «`profile`» w nagłówku przed nazwą pakietu. Posiada on zbiór stereotypów, których atrybuty (nazywane wartościami etykiet definicji) rozszerzają definicję jednej lub większej ilości metaklas (encji). Rozszerzenie polega na dowiązaniu dodatkowych informacji do elementów modelu przechowywanych w repozytorium.

Profil może zostać użyty przez relację dostępu albo importu pakietu. Przykładowo, jeżeli model ma przyjmować określony profil, najwyższy (w hierarchii) pakiet modelu powinien mieć dostęp albo importować pakiet odnoszący się do definicji wymaganego profilu.

Rozszerzenie (*extension*) — relacja pomiędzy stereotypem a klasą metamodelu wskazująca, że stereotyp rozszerza klasę metamodelu. Linia relacji rozszerzenia definiuje tryb dołączania dodatkowej informacji. Jest to określone przez zawartość powiązanego z nią pola tekstowego. Jeżeli występuje w nim wartość 1 to wobec wszystkich elementów, które są instancjami rozszerzanej metaklasy zostanie automatycznie zastosowany stereotyp. Jeżeli występuje wyrażenie `0..1`, to należy ręcznie przypisać stereotyp.

Symbole (klasy, sygnału, itp) mogą wskazywać, że został wobec nich zastosowany ręcznie stereotyp.

Modelowanie wyżej przedstawionych mechanizmów w sieci Petriego może ją znacznie skomplikować. Rozsądne wydaje się, więc postępowanie analogiczne do zdefiniowanego dla klasy abstrakcyjnej — sieć wynikową uzyskuje się po dojściu do najniższego poziomu abstrakcji (modelu). W rezultacie sieć uwzględnia wszystkie modyfikacje związane z mechanizmami obiektowymi.

7.2. Klasy metamodelu

Poniżej zostaną opisane niektóre z ważniejszych klas metamodelu. Ich obecność w niniejszej pracy związana jest z usystematyzowaniem użytych wcześniej pojęć. Konwersję na sieć Petriego w tym przypadku realizuje się w sposób analogiczny do przedstawionego w punkcie 7.1 — Rozszerzalność.

Sygnatura (*signature*) jest metaklasą w języku UML. Jest to element, który może być bazą dla definicji innej sygnatury. Jej wykorzystanie związane jest z jednym z mechanizmów:

– specjalizacji (dziedziczenia) — nadsygnatura może być specjalizowana w zbiór podsygnatur. Każda podsygnatura posiada wszystkie właściwości nadsygnatury i może zawierać dodatkowe. W metamodelu mechanizm specjalizacji jest realizowany przez klasę uogólnioną, którą posiada sygnatura.

– parametryzacji — sygnatura może posiadać listę formalnych parametrów kontekstowych. Jest ona wówczas nazywana szablonem. Jego formalne parametry kontekstowe mogą zostać zastąpione przez rzeczywiste parametry kontekstowe, gdy szablon jest powoływany do życia (na przykład w `TemplateTypeInstantiation`). Parametryzacja może spowodować, że sygnatura będzie bardziej elastyczna w użyciu dla różnych kontekstów. W metamodelu mechanizm parametryzacji jest realizowany przez klasę `ContextParameter`, którą posiada sygnatura.

Oprócz wspomnianych mechanizmów definiowania nowych sygnatur bazujących na innych sygnaturach, występuje trzeci, który posiada tylko jedna sygnatura — synonim. Ten mechanizm definiuje nową sygnaturę przez możliwe zawężenie innej.

Niektóre sygnatury mogą posiadać implementację. W takim przypadku sygnatura działa jako fasada dla implementacji, ukrywając wszystkie szczegóły o których korzystający z sygnatury nie muszą wiedzieć. Fasada umożliwia oddzielenie definicji od jej implementacji a przez to umożliwia osobną kompilację fragmentów systemu (np. jak dla plików nagłówkowych w C). Następujące stwierdzenia są prawdziwe dla fasady:

– fasada nie zależy od jej implementacji

– fasada nie zależy od jej użyć (to jest prawdziwe generalnie dla wszystkich definicji)

– jedynie implementacja może zależeć od fasady

Następujące elementy modelu są sygnaturami:

– klasyfikator (*classifier*) — metaklasa w języku UML. Klasyfikator jest opisem danych oraz sygnaturą zbioru instancji albo instancji zbiorów. Klasyfikator określa typ, który może być na przykład typem strukturalnej cechy. Klasyfikator może być powiązany z innymi klasyfikatorami przy pomocy relacji powiązania.

  Klasyfikatorem jest w większości przypadków element mający cechy klasy, czyli:

  – klasa
  – typ danych, synonim, wybór
  – stereotyp
  – interfejs
  – współpraca

– metoda, sygnał, budzik

Implementacja (*implementation*) opisuje szczegóły sygnatury, których użytkownicy sygnatur nie muszą znać, ale które są konieczne z punktu widzenia realizacji. Sygnatura opisuje zazwyczaj statyczne właściwości encji, natomiast związana z nią implementacja koncentruje się na dynamicznych aspektach. Występują 2 główne rodzaje implementacji: zawartość (*internals*) oraz metoda (*method*). Zawartość opisuje strukturę klasy: fizyczną oraz z punktu widzenia komunikacji, natomiast metoda opisuje operację, typ stanu, albo klasę z dynamicznego punktu widzenia.

Metoda jest implementacją operacji — opisuje sposób działania. Występują 3 rodzaje metod każda z nich ma swoją semantykę wykonania:

– ciało operacji — metoda bezstanowa, której działanie jest realizowane przez wykonywanie akcji `OperationBody`.

– implementacja maszyny stanowej — metoda ze stanami oraz przejściami, której działanie jest realizowane przez wykonywanie akcji powiązanych z przejściami.

– interakcja — metoda, która opisuje interakcje i wymianę informacji pomiędzy zbiorami atrybutów. W przeciwieństwie do innych metod może nie tylko zapewniać całkowitą specyfikację tego jak metoda ma być wykonywana, ale może być również użyta do opisania jak ona w rzeczywistości została wykonana (przez opisanie ścieżki), lub umożliwiając częściowy opis sposobu, w jaki musi zostać wykonana (przez zastrzeżenie wymagań semantycznych odnoszących się do jej innych metod).

Implementacja aktywności — metoda wykonująca kontrolowany zbiór małych jednostek zachowania.

Sygnatura oraz implementacja są metaklasami w języku UML. Sygnatura deklaruje encję natomiast implementacja definiuje tę encję. Celem jest fizyczne oddzielenie sygnatury od implementacji (podobnie jak pliki nagłówkowe w C czy C++).

Pojęciami dla których jest to możliwe są:

– sygnatura metody oraz: ciało operacji, implementacja aktywności, implementacja maszyny stanowej, albo interakcja,

– sygnatura aktywności oraz implementacja aktywności,

– sygnatura maszyny stanowej oraz implementacja maszyny stanowej,

– sygnatura klasy oraz zawartość.

8. Podsumowanie

Pierwotnym celem podjętych badań było opracowanie metody translacji wybranych diagramów języka UML do modelu w postaci kolorowanej sieci Petriego. Translacja taka nie tylko dostarcza formalnej semantyki dla modeli opisanych w języku UML, ale również pozwala na foralną weryfikację modelu z użyciem narzędzi wspierających projektowanie i analizę z użyciem CP-sieci (np. CPN Tools).

Sieć Petriego umożliwia sprawdzanie właściwości dynamicznych projektowanego systemu, których weryfikacja w UML może okazać się problematyczne (nieograniczona symulacja). Jednym z ważniejszych mechanizmów sprawdzania właściwości sieci jest konstrukcja drzew/grafów osiągalności oraz pokrycia. Pierwsze z nich reprezentuje wszystkie możliwe do osiągnięcia stany systemu. Jeżeli sieć może się wykonywać bez końca i nie powraca do wcześniej osiągniętego stanu to graf osiągalności jest nieskończony. W takim przypadku można zbudować graf pokrycia, w którym możliwe jest symboliczne zapisanie stanów, w których liczba znaczników dąży do nieskończoności nie zmieniając przy tym zachowania sieci. Pozwala to na uzyskanie grafu skończonego.

Stan sieci czasowej związany jest między innymi z aktualną wartością zegara. W konsekwencji nie występują jednakowe stany w grafie osiągalności, pomimo nie zwiększającej się liczby znaczników. W takiej sytuacji można zdefiniować relacje równoważności wskazujące, że niektóre stany w pewnych chwilach czasowych są sobie równoważne — nie wnoszą nowej informacji o zachowaniu sieci.

Różnica między grafem pokrycia sieci czasowych oraz nieczasowych wynika przede wszystkim z możliwości zmiany sekwencji odpaleń związanej, z zależnościami czasowymi. Można, więc analizować sieć czasową rozbijając ją na sieci nieczasowe. Rozbicie określa przypadki zależności czasowych, które mają wpływ na sekwencję odpaleń. Ponieważ jest to dodatkowy wymiar, można go zapisać jako kolejną zmienną rozróżniającą wierzchołki nieczasowego drzewa pokrycia. W wyniku otrzymuje się pojedyncze drzewo pokrycia. Innym rozwiązaniem jest zastosowanie modelu czasowego zaproponowanego w [47]. Zamiast globalnego zegara, którego stan zmienia się dążąc do nieskończoności znaczniki otrzymują pieczątki czasowe, których wartości zmniejszają się przy kolejnych cyklach zegara. Przy takim rozwiązaniu możliwe jest znaczne uproszczenie procesu znajdywania równoważnych węzłów w grafie pokrycia.

Analizując sieć Petriego powstałą w wyniku tłumaczenia diagramu UML łatwo jest zidentyfikować fragmenty, w których może nastąpić zróżnicowanie zachowania ze względu na czas. Wynika to z faktu, że właściwie jedynie konstrukcja związana z przeterminowaniem jest jawnym określeniem

zależności czasowych w UML. Również zależności przedstawione na zdefiniowanym w kolejnych wersjach standardu UML diagramie zależności czasowych (*timing diagram*) można modelować analogicznie jak konstrukcję reprezentującą przeterminowanie.

Graf osiągalności/pokrycia przedstawia całe zachowanie systemu pozwalając na analizę wszystkich możliwych przypadków. Pozwala to na wykrycie stanów, które nie zostały przewidziane w procesie projektowania przez co mogą powodować błędne działanie systemu.

Innym mechanizmem sprawdzania właściwości sieci Petriego jest wyznaczanie niezmienników. Przedstawiają one fragment sieci w którym liczba znaczników nie zmienia się. Można to interpretować jako rodzaj autonomiczności działania fragmentu w stosunku do pozostałej części sieci. Przedstawiony fragment jest więc niezależny od poprawności działania pozostałych elementów systemu.

Wykorzystanie sieci Petriego prowadzi do wyznaczenia pewnych cech projektowanego oprogramowania. Jeżeli ujawni się cecha niepożądana, to należy zidentyfikować fragment modelu który jest za to odpowiedzialny. Wydaje się, że najprostszym sposobem powrotu z reprezentacji w sieci Petriego na diagram UML jest wykorzystanie oznaczonych podczas tłumaczenia fragmentów. Oznaczanie sprowadza się do wskazania, że dany element diagramu UML został przetłumaczony na wskazany fragment sieci Petriego. Interpretacja wyników analizy sieci Petriego wykorzystuje przedstawione wyżej informacje w drugą stronę wyznacza element(y) diagramu UML, który odpowiada za otrzymany rezultat.

Konwersja odwrotna (sieci Petriego $\rightarrow$ UML) może zostać zrealizowana z wykorzystaniem informacji uzyskanych podczas pierwotnego etapu konwersji. W tym celu należy podczas konwersji oznaczać fragmenty źródłowe oraz odpowiadające im konstrukcje wynikowe.

Dokładne modelowanie mechanizmów wykorzystywanych w systemie umożliwia uwzględnienie czasu wykonywania poszczególnych konstrukcji pozwalając na zbudowanie modelu, który umożliwi formalną analizę pod kątem zachowań zależnych od czasu.

8.1. Wnioski końcowe

Za najistotniejsze wyniki zawarte w pracy należy więc uznać:

- Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu sekwencji oraz ogólnego diagramu interakcji.
- Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu stanów oraz diagramu przepływu.
- Opracowanie algorytmu generowania sieci Petriego na podstawie diagramu klas, diagramu architektury oraz diagramu stanów.
- Stworzenie sieci Petriego realizującej przesyłanie sygnałów zgodnie z mechanizmmi UML.
- Rozbudowanie generowanej sieci o mechanizmy umożliwiające sprawdzenie poprawności działania w aspektach nie analizowanych z poziomu UML.

– Opracowanie przykładu ilustrującego przydatność przedstawionych metod konwersji.

8.2. Perspektywy dalszych badań

Kolejny etapem badań nad rozwijaniem opracowanych rozwiązań będzie próba stworzenia algorytmu porównującego sieci Petriego utworzone na podstawie diagramów z różnych etapach projektowania oprogramowania. Niniejsza praca prezentuje algorytmy tworzenia sieci Petriego na 3 różnych poziomach ogólności. Pozwala to na formalną weryfikację każdego z tych etapów. Nie ma jednak pewności, że kolejne poziomy uszczegóławiania modelu odpowiadają wymaganiom stawianym na wcześniejszych etapach. Dalszy rozwój badań będzie zatem ukierunkowany na określenie zależności pomiędzy modelami stworzonymi na podstawie różnych etapów projektowania oprogramowania.

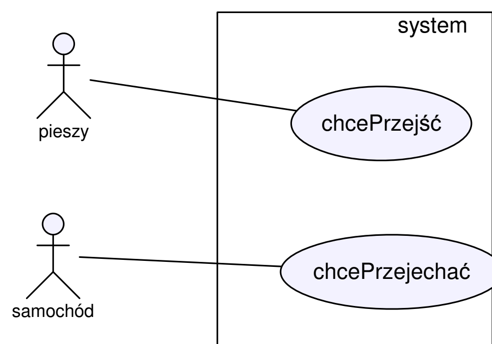
8.3. Podziękowania

Serdecznie dziękuję Panu Prof. dr hab. Marcinowi Szpyrce za pomoc, bez której niniejsza praca nie zostałaby ukończona.

A. Dodatek

Poniżej przedstawiono przykład systemu opisanego przy pomocy UML. Jest to model sterownika sygnalizacji świetlnej skrzyżowania ciągu pieszego z drogą. Nadchodzący pieszy generuje sygnał „chcePrzejść” w celu zmiany wyświetlanego sygnału na zezwalający na przejście. Sygnał powoduje sekwencyjną zmianę stanu sygnalizacji, aby umożliwić bezpieczną zmianę kierunków ruchu. Kolejna zmiana stanu sygnalizacji może nastąpić po otrzymaniu przez sterownik sygnału „chcePrzejechać”. W tym przypadku sekwencja zmian powinna doprowadzić sygnalizację do stanu z początku niniejszego opisu. Jest to jeden z prostszych przykładów oprogramowania sterownika sygnalizacji świetlnej. Ideą przykładu jest pokazanie możliwych sposobów wykorzystania sieci Petriego w procesie tworzenia oprogramowania. Przedstawiona jest translacja dla 2 etapów projektowania oprogramowania. Dla zapewnienia czytelności oraz zwięzłej formy pracy przedstawiony przykład koncentruje się na zaprezentowaniu translacji konstrukcji wymaganych dla prawidłowego odwzorowania specyfiki UML.

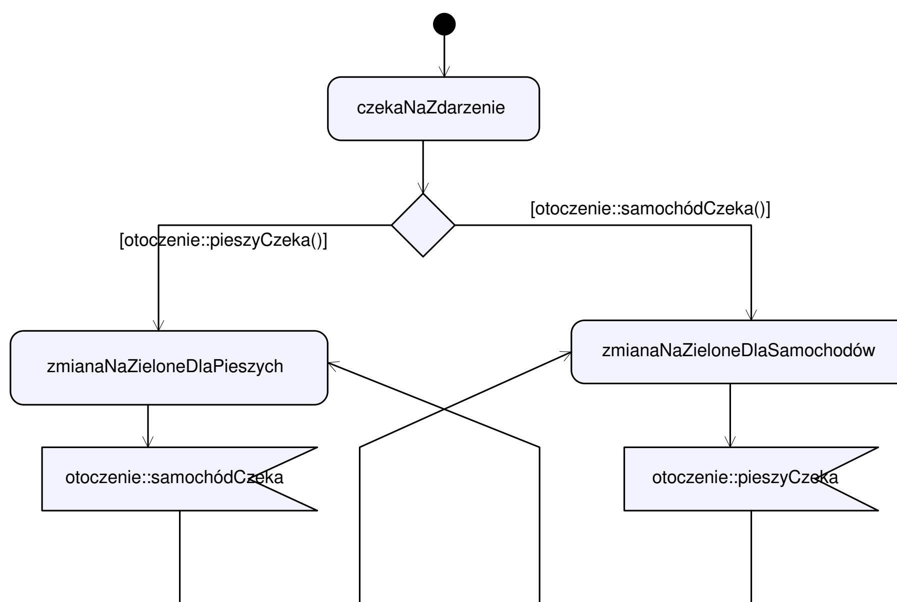
Najbardziej ogólna definicja systemu przedstawiona jest na diagramie przypadków użycia (patrz rysunek A.1). Definiuje on granicę pomiędzy systemem a otoczeniem. W omawianym przykładzie system jest sterownikiem sygnalizacji, wykorzystywanej przez pieszych oraz pojazdy — zdefiniowani jako aktorzy. Aktorzy mogą wpływać na system, aby sterownik wyświetlił sygnał umożliwiający przejście/przejazd. Sygnał powinien zostać obsłużony przez przypisaną mu odpowiednią funkcjonalność systemu.



Rysunek A.1: Diagram przypadków użycia sygnalizacji świetlnej

Jak wspomniano przy omawianiu algorytmu konwersji na tym etapie tworzenia systemu informacje są zbyt ogólne, aby można je było wykorzystać do utworzenia sieci Petriego.

Dokładniejsze odwzorowanie działania sterownika przedstawia diagram aktywności (patrz rysunek A.2). Znajdują się na nim 2 aktywności związane ze zmianą świateł, które są przełączane sygnałami o nadejściu pieszego lub nadjechaniu pojazdu.



Rysunek A.2: Diagram aktywności dla systemu sterowania ruchem

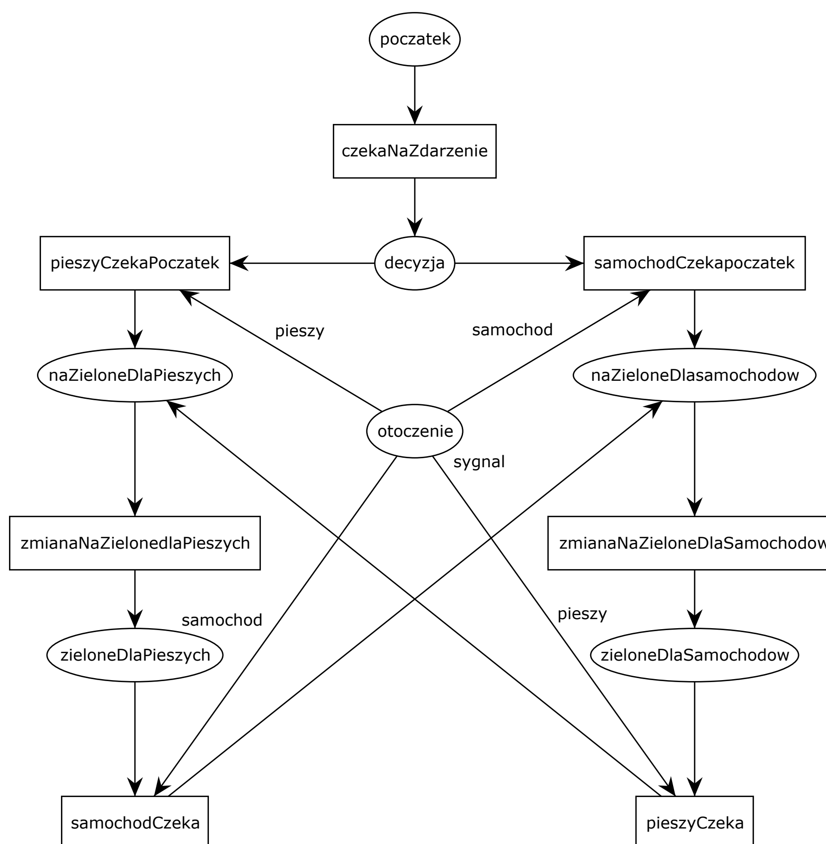
W wyniku konwersji opisywanego diagramu powstaje sieć Petriego przedstawiona na rysunku A.3. Jak można zauważyć sieć zawiera więcej elementów niż diagram. Wiąże się to z koniecznością zapewnienia dwudzielności grafu.

W dalszej części znajduje się opis najbardziej szczegółowej reprezentacji aplikacji.

Diagram pakietu przedstawia definicje elementów, relacje zachodzące pomiędzy nimi oraz interfejsy (patrz rysunek A.4). Interfejsy określają jaka funkcjonalność jest wymagana oraz realizowana. Definicje interfejsów oraz związanych z nimi sygnałów znajdują się na diagramie. Ze względu na organizacyjny charakter diagramu nie ma on bezpośredniej reprezentacji w postaci sieci Petriego. Informacje do budowy sieci w nim zawarte można wydobyć z pozostałych diagramów.

Diagram klas (patrz rysunek A.5) przedstawia relacje pomiędzy definicjami wykorzystywanymi w aplikacji. Są na nim również zaznaczone obiekty, które zostaną utworzone podczas działania aplikacji. Na podstawie tego diagramu generowana jest sieć Petriego określająca kolejność wywoływania konstruktorów (patrz rysunek A.6). Ze względu na pojedyncze instancje każdego z obiektów nie jest konieczne definiowanie konstrukcji związanej z identyfikacją instancji. W związku z tym działanie konstruktorów sprowadza się do aktywacji kolejek sygnałów dla każdego z obiektów. Kolejną informacją istotną dla budowy sieci jest nazwa portu, przez który następuje komunikacja z otoczeniem.

Definicje zachowań obiektów znajdują się na diagramach stanów. Pierwszy z nich przedstawia maszynę stanową sterownika sygnalizacji. Jego działanie zaczyna się od oczekiwania na sygnał nadejścia pieszego lub nadjechania samochodu. W zależności od pierwszego sygnału wybierana jest odpowiednia gałąź z której wysyłany jest do sygnalizatora sygnał zmiany światła na zielone. Zmiana



Rysunek A.3: Sieć wynikowa po translacji diagramu aktywności UML

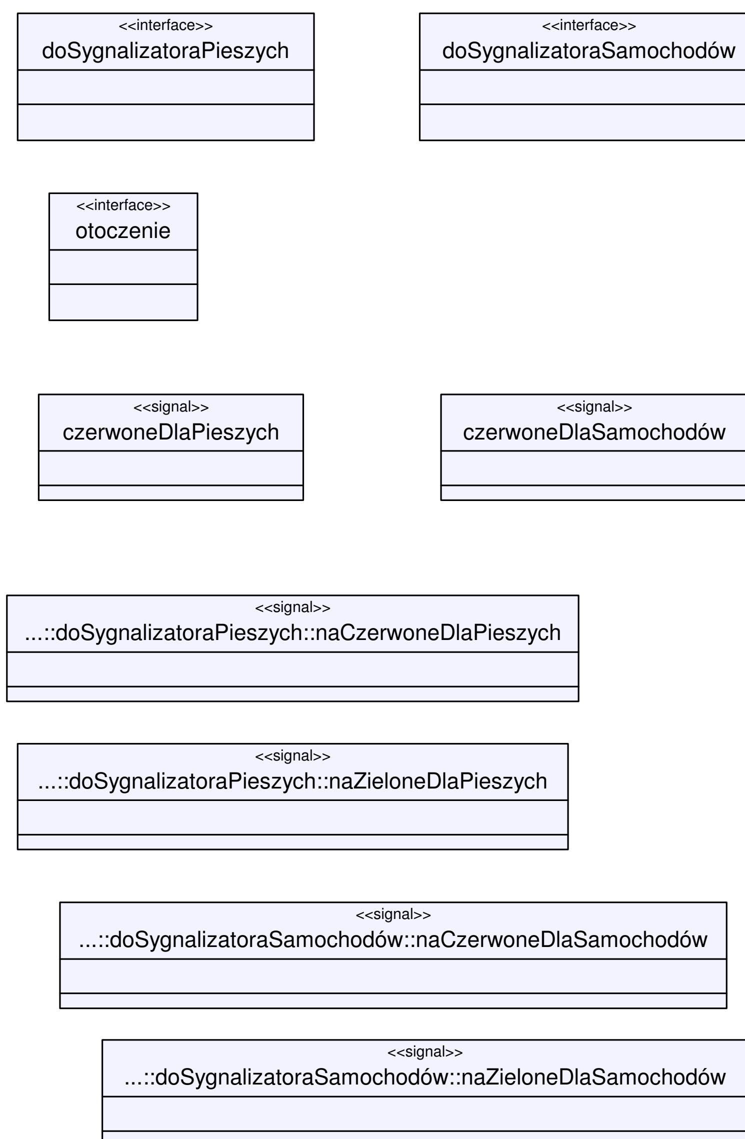
stanu sterownika możliwa jest po otrzymaniu sygnału oczekiwania z kierunku prostopadłego. Wówczas sterownik wysyła sygnał zmiany na czerwone i czeka na potwierdzenie tej zmiany. Następnie wysyłany jest sygnał zmiany na zielone do drugiego sygnalizatora.

Tworzenie sieci Petriego dla maszyny stanowej zaczyna się od zdefiniowania kolejki sygnałów (patrz rysunek A.7). Elementy pobrane z kolejki sygnałów mogą być odebrane, jeżeli maszyna jest w odpowiednim stanie. Takie zachowanie zapewnia konstrukcja przedstawiona na rysunku A.8. Uniemożliwia ona obsłużenie jednorazowo więcej niż jednego sygnału.

Dla poprawnego działania poniższych sieci wymagane jest utworzenie następujących deklaracji:

```
colset sterowanie=unit;
colset odbiorca=int with 1..7;
var od, odc: odbiorca;
colset sygnaly=string timed;
var syk, sy, syc: sygnaly;
colset sygnal=record syg: sygnaly*odb: odbiorca;
colset lista=list sygnaly;
var l: lista;
```

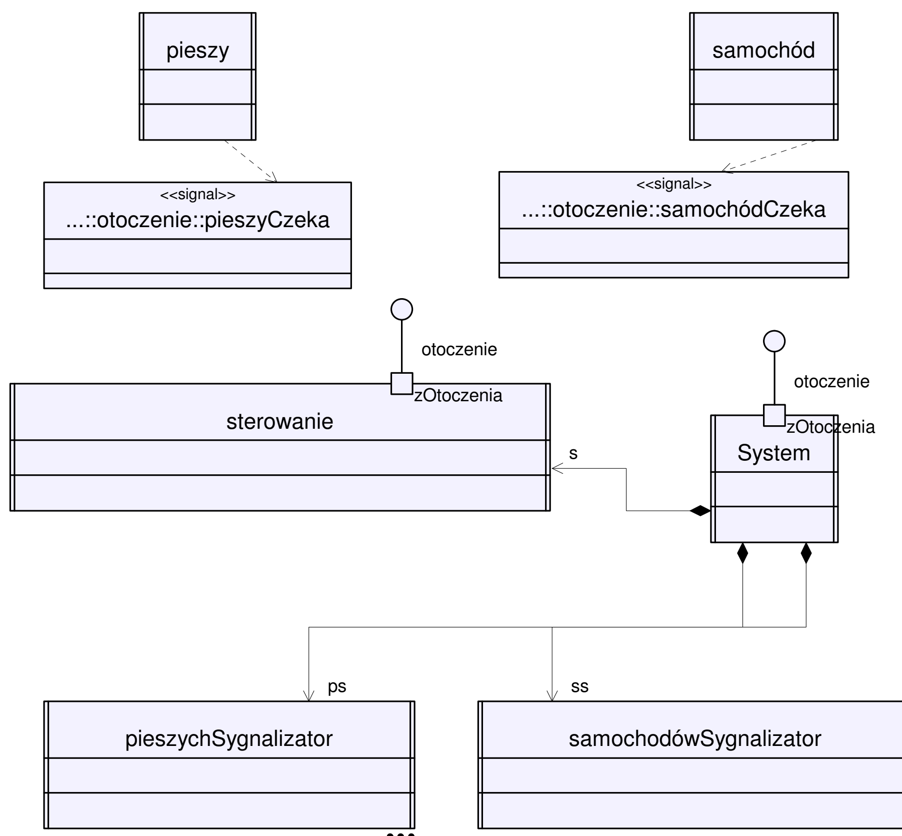
Następnie tworzona jest strona z tłumaczeniem poszczególnych konstrukcji UML na sieci Petriego. W omawianym przypadku ze względu na skomplikowanie maszyny stanowej, sieć Petriego znajduje się na dwóch stronach (oprócz sieci związanej z obsługą sygnałów).



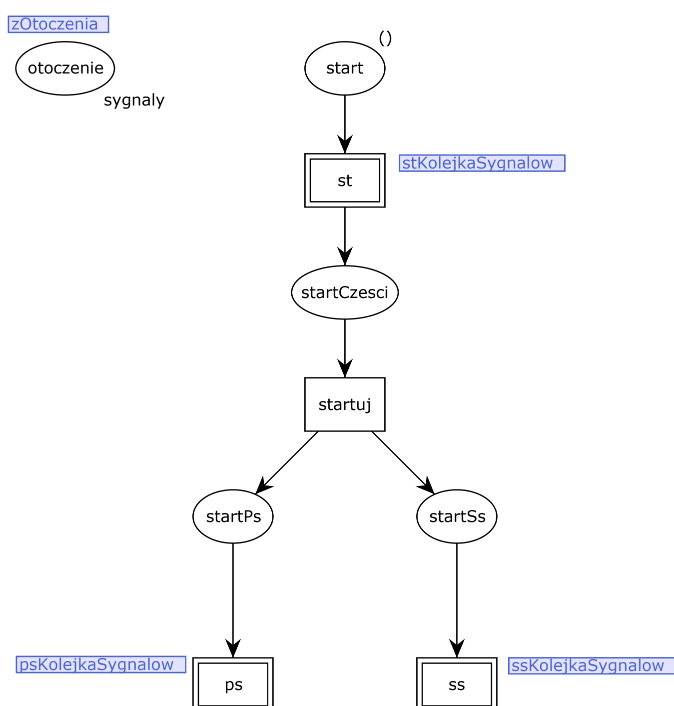
Rysunek A.4: Diagram pakietu

Ze względu na automatyczne pobieranie sygnału o nadjechaniu pojazdu jak i możliwość wielokrotnego wysłania sygnału od pieszego system powinien poradzić sobie ze zwiększoną liczebnością sygnałów. Należy tu zwrócić uwagę na inne zachowanie sieci, która zostałaby zbudowana aby zamodelować działanie systemu bez zagłębiania się w szczegóły implementacyjne. W przypadku sieci zbudowanej na podstawie modelu UML sygnały wysyłane są do kolejki związanej z modelowaną maszyną stanową. Jeżeli maszyna znajduje się w stanie, w którym nie może obsłużyć sygnału jest on odrzucany. W wersji zamodelowanej od początku przy pomocy sieci Petriego mechanizm ten zostałby zaimplementowany niezależnie dla sygnałów od pieszych oraz samochodów.

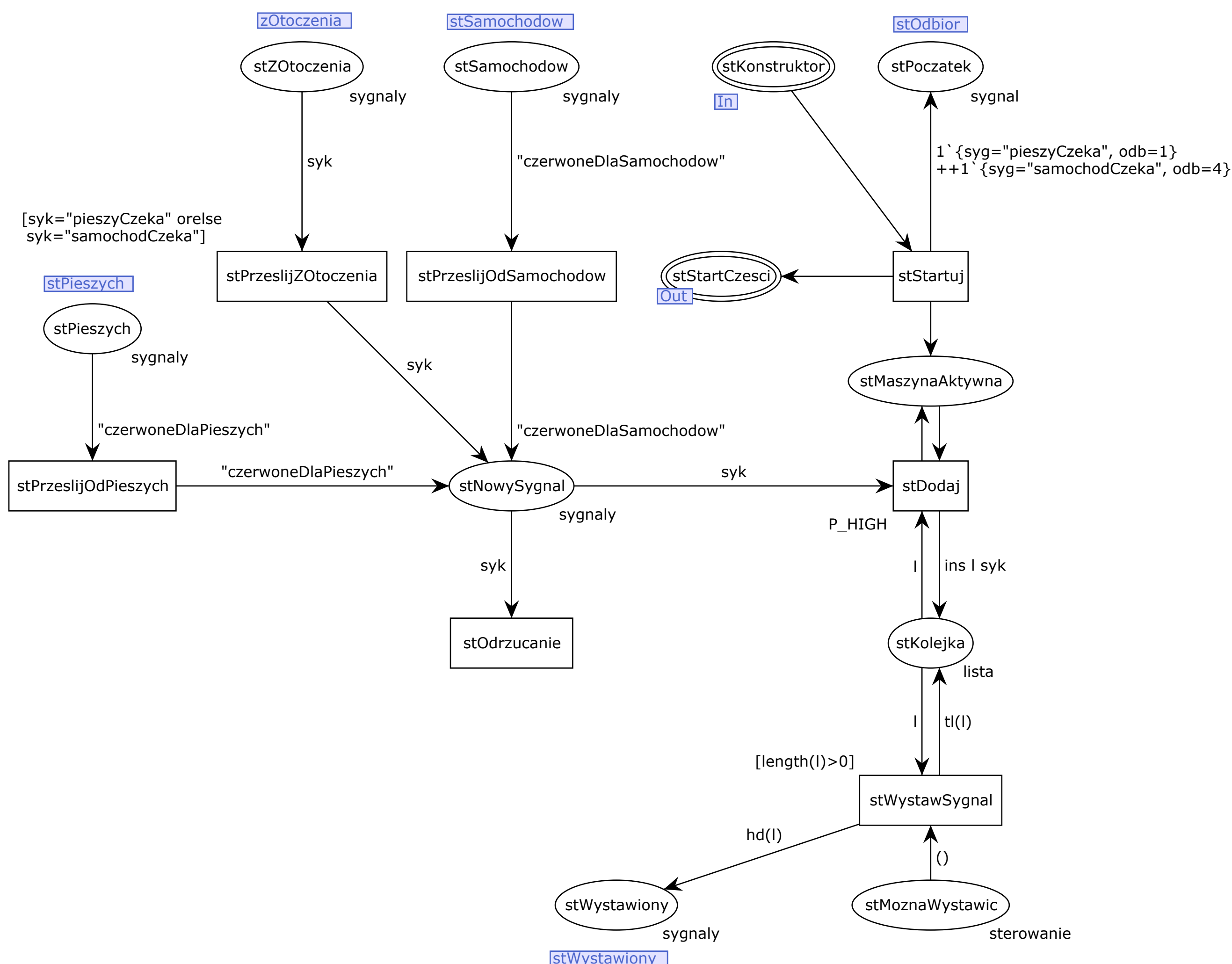
Kolejne diagramy stanów przedstawiają działanie sygnalizatorów dla samochodów (patrz rysunek A.9) oraz dla pieszych.W obu przypadkach maszyny startują od stanu odpowiadającemu światłu czerwonemu. Po otrzymaniu sygnału do zmiany światła od sterownika następuje zmiana stanu wraz z ustawieniem budzika, który gwarantuje odpowiedni czas świecenia. Zmiana z światła zielonego



Rysunek A.5: Diagram klas sterownika sygnalizacji świetlnej



Rysunek A.6: Sieć nadrzędna, która uruchamia „maszyny stanów”



Rysunek A.7: Sieć reprezentująca kolejkę sygnalów dla maszyny stanowej sterownika

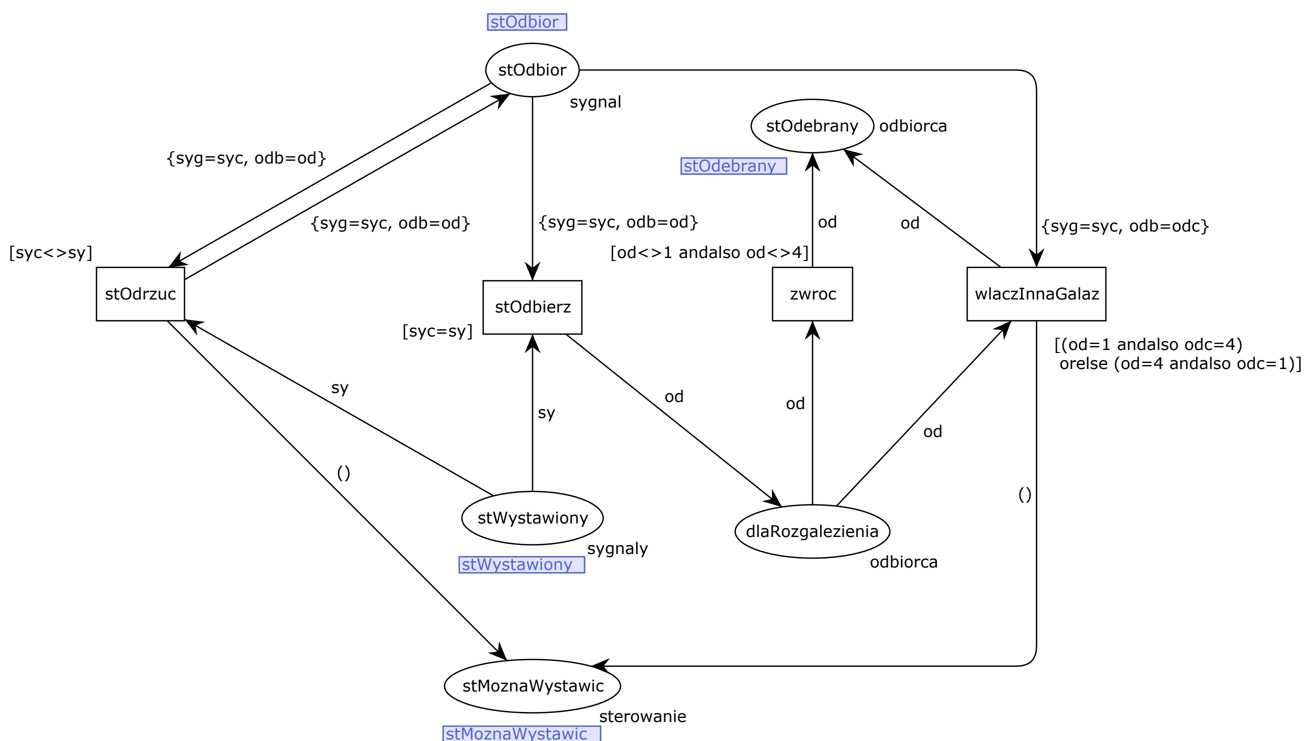
może nastąpić gdy maszyna otrzyma sygnał od sterownika. Aby zapewnić minimalny czas świecenia przed stanem reprezentującym światło zielone ustawiany jest budzik.

Analogicznie jak w przypadku wcześniej omawianej maszyny stanowej sieć Petriego odpowiadająca powyższym diagramom będzie zawierać konstrukcje związane z obsługą sygnałów (rysunki A.10, A.11, A.12, A.13). Jak widać na rysunkach obsługa sygnałów może być wspólną siecią dla wielu maszyn stanowych. Należy wówczas zapewnić odpowiednie rozróżnienie, aby nadchodzące sygnały były przesyłane do właściwej maszyny stanowej. W omawianym przykładzie ze względu na czytelność sieć została skopiowana wraz z odpowiednimi modyfikacjami.

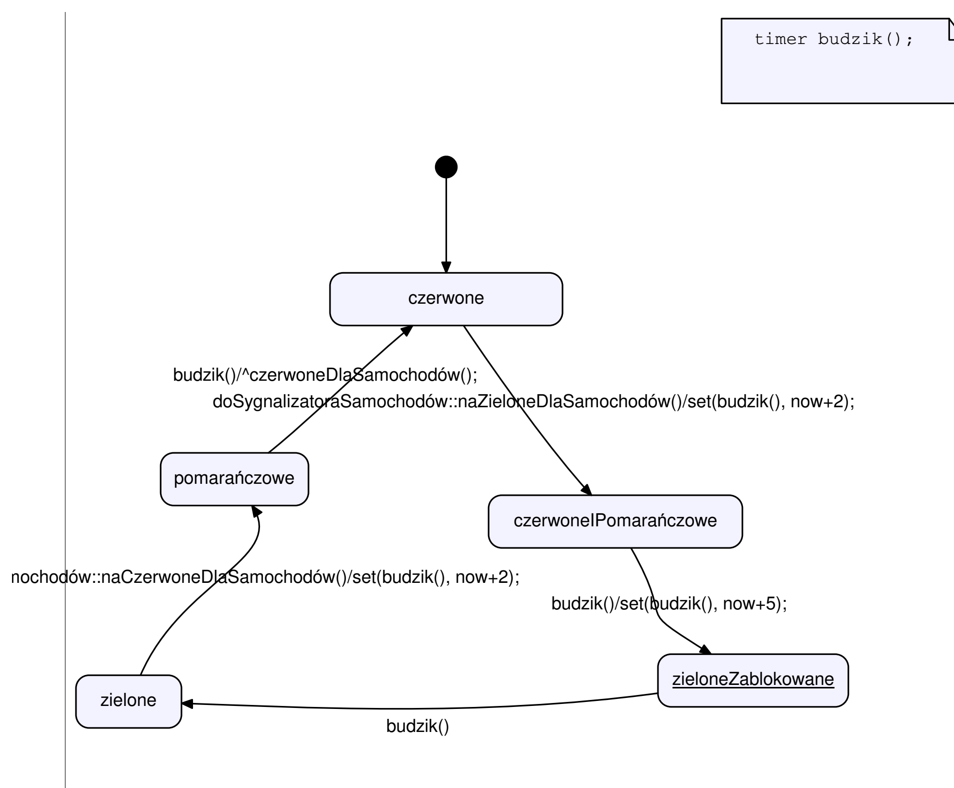
Kolejnym elementem związanym z obsługą sygnałów jest konstrukcja zapisująca sygnały w przypadku, gdy maszyna stanowa nie może ich obsłużyć. Sieci dla poszczególnych maszyn stanowych znajdują się odpowiednio na rysunkach A.14, A.15.

Realizacja algorytmów maszyn stanowych zawarta jest odpowiednio na rysunkach A.16 oraz A.17.

Przedstawiony wyżej przykład obrazuje odwzorowanie diagramów UML w sieci Petriego z wykorzystaniem algorytmu opisanego w niniejszej pracy. Konwersja ma być wykonywana automatycz-

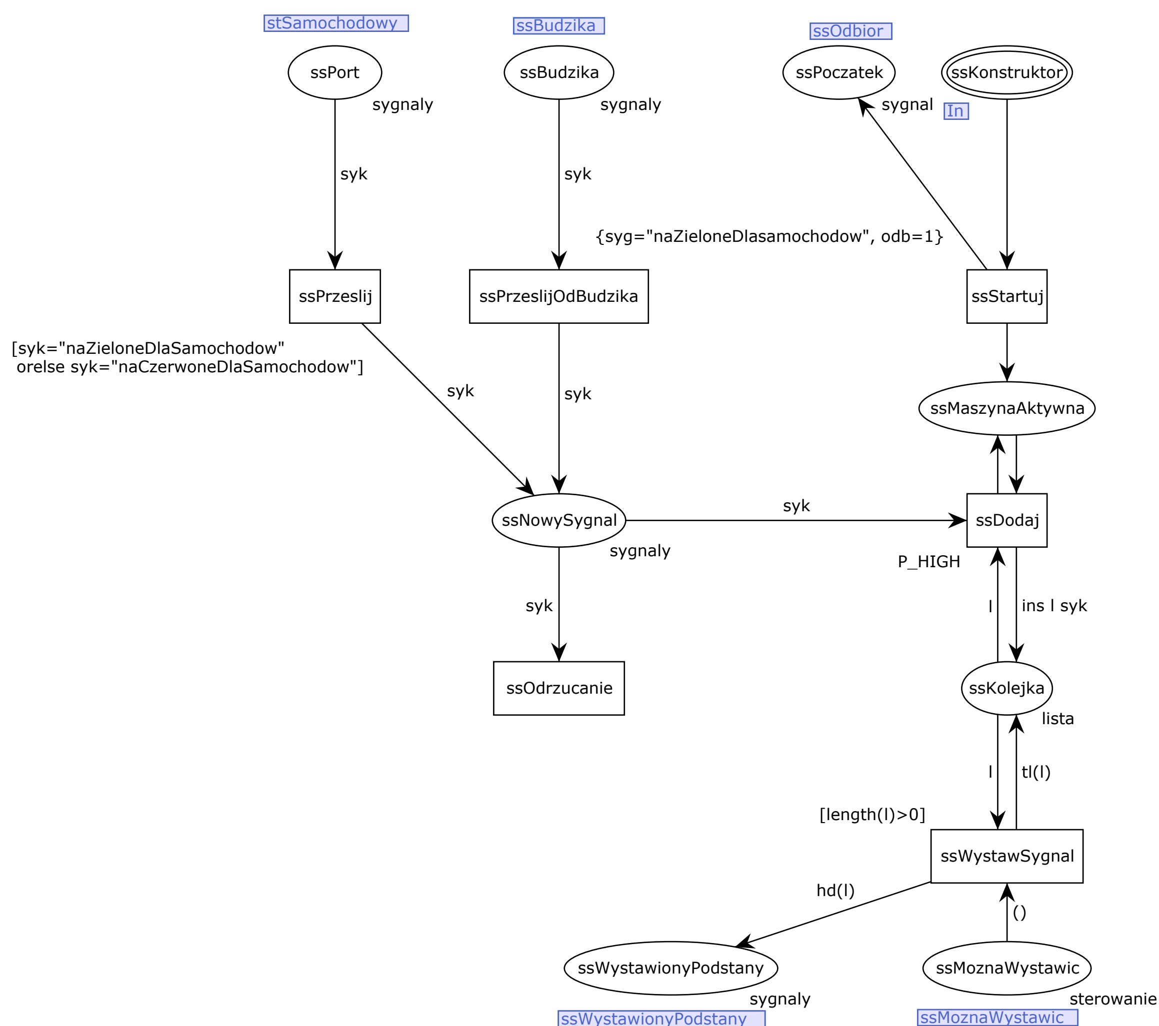


Rysunek A.8: Sieć reprezentująca konstrukcję odbierającą sygnały dla sterownika

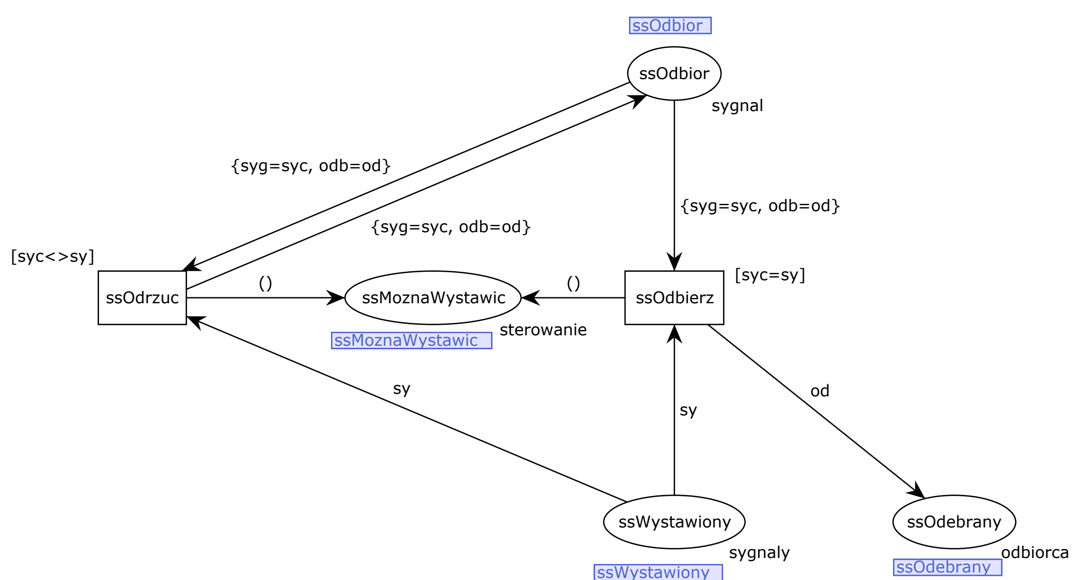


Rysunek A.9: Maszyna stanowa sygnalizatora dla pojazdów

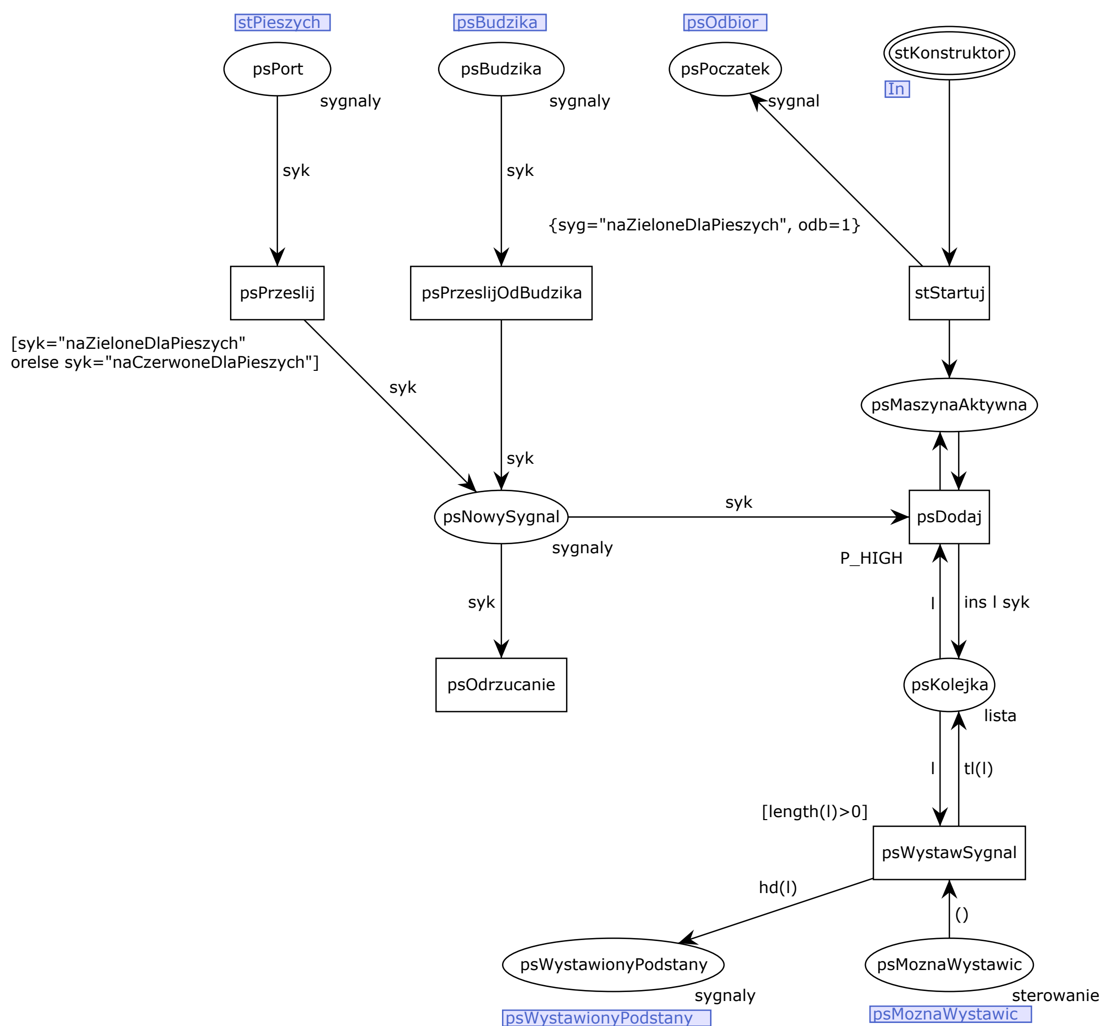
nie po zakończeniu poszczególnych etapów projektowania, więc nie będzie wymagała dodatkowej ingerencji programisty.



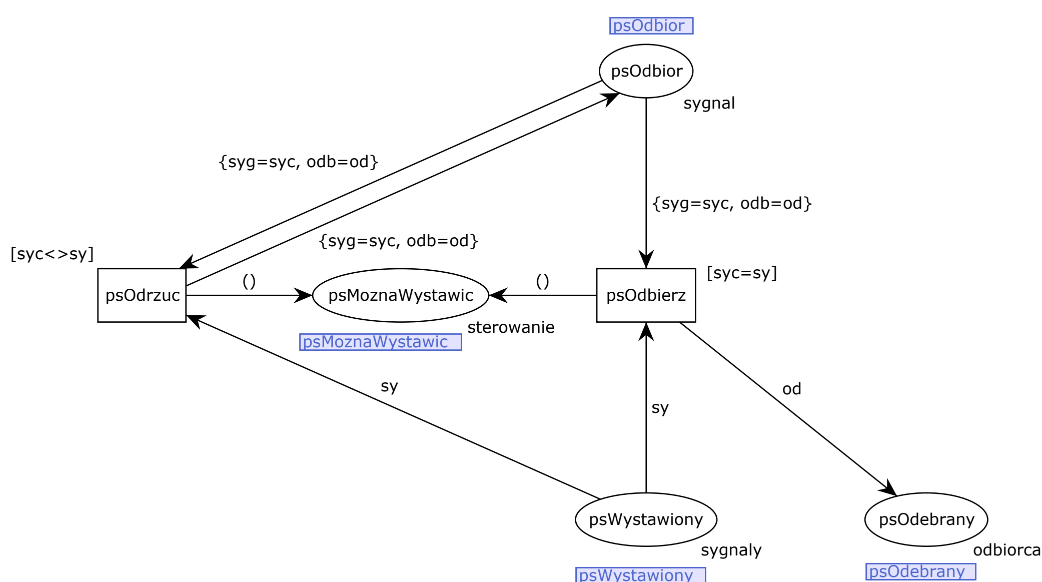
Rysunek A.10: Sieć Petriego realizująca kolejkę sygnałów dla maszyny stanowej sygnalizatora samochodowego



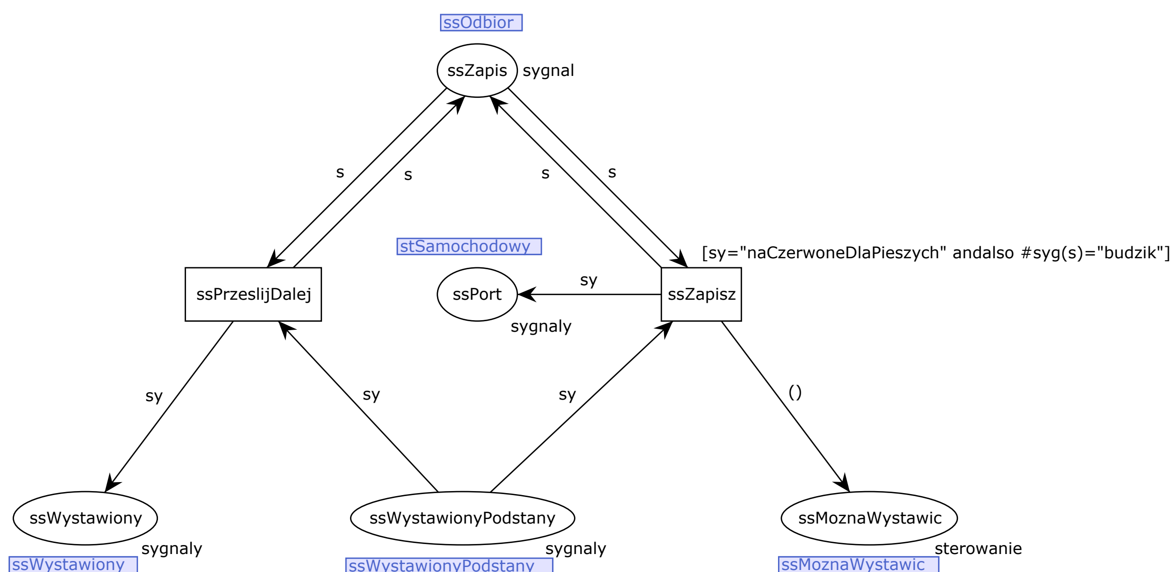
Rysunek A.11: Sieć Petriego reperentująca konstrukcję odbioru sygnału dla sygnalizatora samochodowego



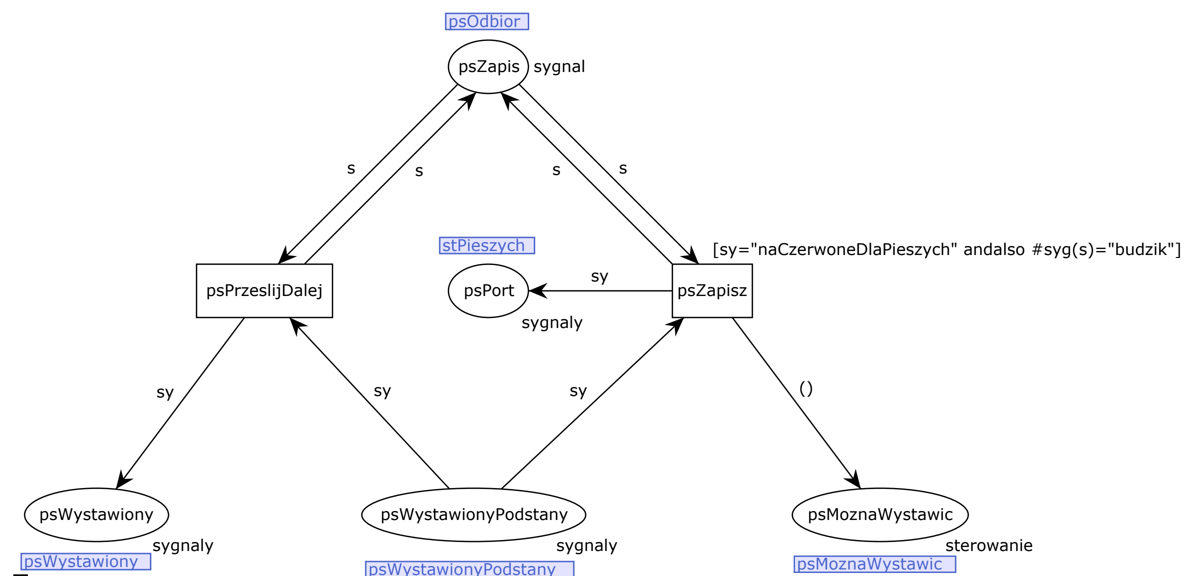
Rysunek A.12: Sieć Petriego realizująca kolejkę sygnałów maszyny stanowej sygnalizatora dla pieszych



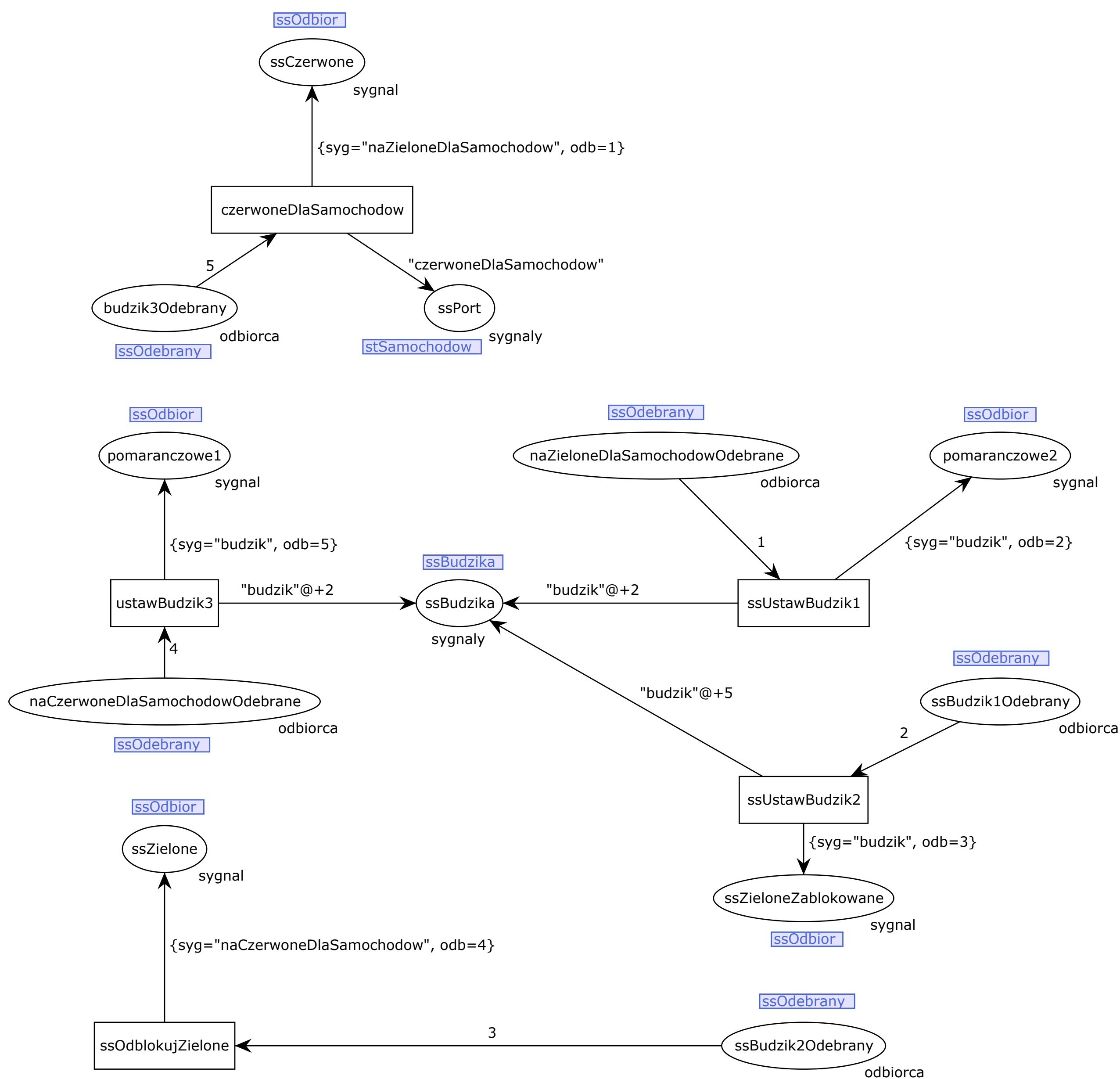
Rysunek A.13: Sieć Petriego reperentująca konstrukcję odbioru sygnału sygnalizatora dla pieszych



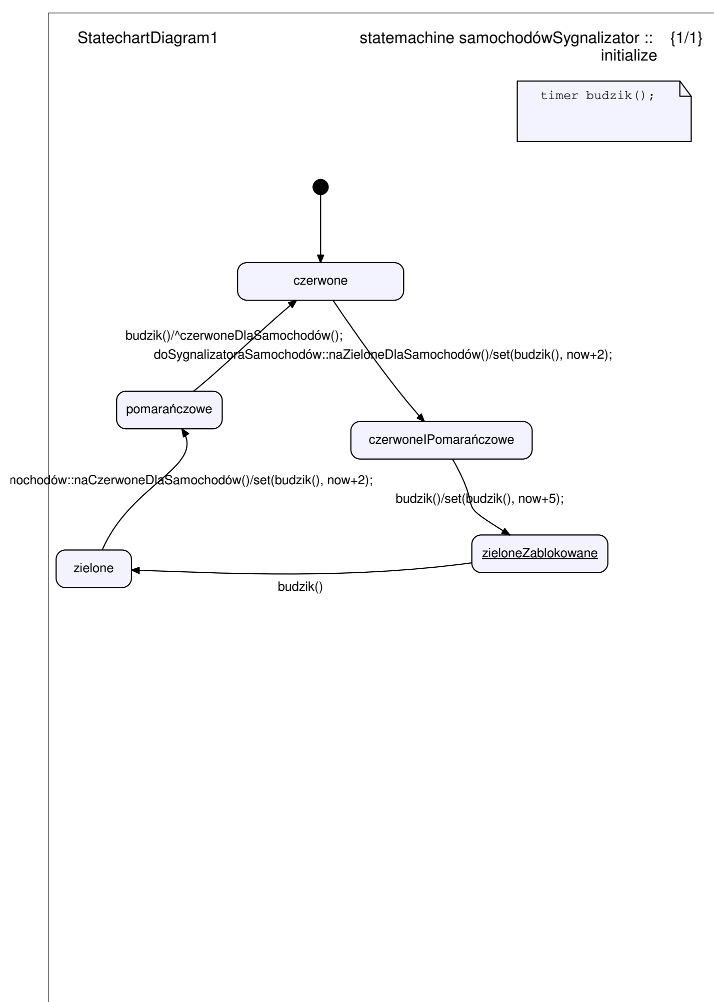
Rysunek A.14: Sieć Petriego przedstawiająca konstrukcję zapisywania sygnałów sygnalizatora dla pojazdów



Rysunek A.15: Sieć Petriego przedstawiająca konstrukcję zapisywania sygnałów sygnalizatora dla pieszych



Rysunek A.16: Sieć Petriego realizująca funkcjonalność maszyny stanowej sygnalizatora samochodowego



Rysunek A.17: Sieć Petriego realizująca funkcjonalność maszyny stanowej sygnalizatora dla pieszych

Bibliografia

[1] Alur R., Dill D.: *Automata for modelling real-time systems*. [In:] *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP'90)*. Vol. 443, LNCS. Springer-Verlag, 1994, 322–335

[2] Alur R., Dill D.: *A theory of timed automata*. Theoretical Computer Science, vol. 126, no. 2, 1994, 183–235

[3] Baier C., Katoen J.-P.: *Principles of Model Checking*. London, UK, The MIT Press, 2008

[4] Baresi L., Pezze M.: *On Formalizing UML with High-Level Petri Nets*. [In:] Agha G., De Cindio F., Rozenberg G., (Eds.): *Concurrent Object-Oriented Programming and Petri Nets*. Vol. 2001, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, 276–304

[5] Bernardi S., Donatelli S., Merseguer J.: *From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models*. [In:] *Proceedings of the 3rd International Workshop on Software and Performance*. WOSP'02, New York, NY, USA, 2002, ACM, 35–45

[6] Berthomieu B., Diaz M.: *Modeling and Verification of Time Dependent Systems Using Time Petri Nets*. IEEE Transactions on Software Engineering, vol. 17, no. 3, 1991, 259–273

[7] Booch G., Rumbaugh J., Jacobson I.: *UML przewodnik użytkownika*. Warszawa, WNT, 2001

[8] Clarke E., Grumberg O., Peled D.: *Model Checking*. Cambridge, Massachusetts, The MIT Press, 1999

[9] David R., Alla H.: *Discrete, Continuous and Hybrid Petri Nets*. Berlin, Germany, Springer-Verlag, 2005

[10] del Bianco V., Lavazza L., Mauri M.: *Model checking UML specifications of real time software*. [In:] *Engineering of Complex Computer Systems, 2002. Proceedings. Eighth IEEE International Conference on*. 2002, 203–212

[11] Dennis A., Wixom B., Tegarden D.: *Systems Analysis and Design with UML*. John Wiley & Sons, 2012

[12] El Miloudi K., El Amrani Y., Ettouhami A.: *An Automated Translation of UML Class Diagrams into a Formal Specification to Detect UML Inconsistences*. [In:] *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA 2011)*. 2011, 432–438

[13] Emerson E.: *Temporal and modal logic*. [In:] van Leeuwen J., (Ed.): *Handbook of Theoretical Computer Science*. Vol. B, Elsevier Science, 1990, 995–1072

[14] Fowler M.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003

[15] Galton A., (Ed.): *Temporal logics and their Applications*. London, Academic Press, 1987

[16] Gherbi A., Khendek F.: *Timed-Automata Semantics and Analysis of UML/SPT Models with Concurrency*. [In:] *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*. 2007, 412–419

[17] Hilderink G. H.: *Graphical modelling language for specifying concurrency based on CSP*. IEE Proceedings Software, vol. 150, 2003, 108–120

[18] Hoare C. A. R.: *Communicating sequential processes*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1985

[19] Holzmann G. J.: *The model checker SPIN*. IEEE Transactions on Software Engineering, vol. 23, no. 5, 1999, 1–17

[20] Jacobs J., Simpson A. On a Process Algebraic Representation of Sequence Diagrams. Department of Computer Science, University of Oxford, 2014

[21] Jensen H. E., Guldstrand K., Skou A.: *Scaling up UPPAAL: automatic verification of real-time systems using compositionality and abstraction*. [In:] *Proc. FTRTFT 2000. 84 ALTISEN ET AL.* 2000

[22] Jensen K.: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Vol. 1–3, Berlin, Germany, Springer-Verlag, 1992-1997

[23] Jensen K., Kristensen L.: *Coloured Petri nets. Modelling and Validation of Concurrent Systems*. Heidelberg, Springer, 2009

[24] Jensen K., Kristensen L., Wells L.: *Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems*. International Journal on Software Tools for Technology Transfer, vol. 9, no. 3–4, 2007, 213–254

[25] Kerkouche E., Chaoui A. A., Bourennane E. B., Labbani O.: *A UML and Colored Petri Nets Integrated Modeling and Analysis Approach using Graph Transformation*. Journal of Object Technology, vol. 9, no. 4, July 2010, 25–43

[26] Klimek R.: *Wprowadzenie do Logiki Temporalnej*. Kraków, AGH Uczelniane Wydawnictwa Naukowo-Dydaktyczne, 1999

[27] Kripke S.: *A semantical analysis of modal logic I: normal modal propositional calculi*. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, vol. 9, 1963, 67–96, (Announced in *Journal of Symbolic Logic*, **24**, 1959, p. 323)

[28] Latella D., Majzik I., Massink M.: *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker*. Formal Aspects of Computing, vol. 11, no. 6, 1999, 637–664

[29] Merz S., Rauh C.: *Model checking timed UML state machines and collaborations*. [In:] *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002*. 2002, 395–414

[30] Murata T.: *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEEE, vol. 77, no. 4, 1989, 541–580

[31] Ober I., Graf S., Ober I.: *Validation of UML Models via a Mapping to Communicating Extended Timed Automata*. [In:] Graf S., Mounier L., (Eds.): *Model Checking Software*. Vol. 2989, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, 127–145

[32] Penczek W., Półrola A.: *Advances in Verification of Time Petri nets and Timed Automata. A Temporal Logic Approach*. Vol. 20, Studies in Computational Intelligence, Springer-Verlag, 2006

[33] Petri C. A.: *Communication with Automata*. Technical report, New York, 1965, Supplement 1 to Technical Report RADC–TR–65–377, (English translation of *Kommunikation mit Automaten*, PhD Dissertation, University of Bonn, 1962)

[34] Rausand M.: *Reliability of Safety-Critical Systems: Theory and Applications*. Wiley, 2014

[35] Redmill F., Rajan J.: *Human factors in safety-critical systems*. Oxford, Reed Educational and Professional Publishing Ltd., 1997

[36] Remenska D., Templon J., Willemse T., Homburg P., Verstoep K., Casajus A., Bal H.: *From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems*. [In:] Brat G., Rungta N., Venet A., (Eds.): *NASA Formal Methods*. Vol. 7871, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, 244–260

[37] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Reference Manual*. Boston, MA, Addison-Wesley, 2 edition, 2005

[38] Sommerville I.: *Software Engineering*. London, Pearson Education Limited, 2004

[39] Storey N.: *Safety-critical computer systems*. Addison-Wesley, 1996

[40] Szlenk M.: *Formalna semantyka i wnioskowanie o pojęciowym diagramie klas w UML*. Praca doktorska, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, 2005. promotor prof. dr hab. inż. K. Sacha

[41] Szmuc T.: *Zaawansowane metody tworzenia oprogramowania systemów czasu rzeczywistego*. Kraków, CCATIE, 1998

[42] Szmuc T., Motet G.: *Specyfikacja i projektowanie oprogramowania systemów czasu rzeczywistego*. Kraków, Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, 2000

[43] Szmuc T., Szpyrka M., (Eds.): *Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego*. Warszawa, WNT, 2010

[44] Szmuc W.: *Wytwarzanie oprogramowania czasu rzeczywistego wspomagane metodą formalną: SDL – kolorowane sieci Petriego*. Automatyka, vol. 7, 2003, 267–273

[45] Szmuc W.: *Hybrid modeling of software systems – UML/Petri nets approach*. [In:] Grzech A., (Ed.): *Proceedings of the 16th international Conference on Systems Science*. Vol. 3, Wrocław, Oficyna Wydawnicza Politechniki Wrocławskiej, 2007, 389–398

[46] Szmuc W.: *Modelowanie konstrukcji obiektowych języka UML z zastosowaniem kolorowych sieci Petriego*. Automatyka, vol. 11, 2007, 287–295

[47] Szpyrka M.: *Analysis of RTCP-nets with Reachability Graphs*. Fundamenta Informaticae, vol. 74, no. 2–3, 2006, 375–390

[48] Szpyrka M.: *Analysis of VME-Bus communication protocol – RTCP-net approach*. Real-Time Systems, vol. 35, no. 1, 2007, 91–108

[49] Szpyrka M.: *Modelowanie i analiza systemów wbudowanych z zastosowaniem RTCP-sieci*. Vol. 165, Rozprawy Monografie, Kraków, AGH Uczelniane Wydawnictwa Naukowo-Dydaktyczne, 2007

[50] Szpyrka M.: *Sieci Petriego w modelowaniu i analizie systemów współbieżnych*. Warszawa, WNT, 2008

[51] Szpyrka M., Biernacka A., Biernacki J.: *Methods of translation of Petri nets to NuSMV language*. [In:] *Proceedings of the Concurrency Specification and Programming Workshop (CSP 2014)*. Vol. 1269, CEUR Workshop Proceedings, Chemnitz, Germany, September 29-October 1 2014, 245–256

[52] Szpyrka M., Szmuc T., Matyasik P., Szmuc W.: *RTCP-sieci - formalne podejcie do szybkiego modelowania systemów czasu rzeczywistego*. [In:] Górski J., Wardziński A., (Eds.): *Inżynieria oprogramowania*. Warszawa, WNT, 2004, chapter 23, 315–328

[53] Szpyrka M., Szmuc T., Matyasik P., Szmuc W.: *A Formal Approach to Modelling of Real-time Systems Using RTCP-nets*. Foundations of Computing and Decision Sciences, vol. 30, no. 1, 2005, 61–71

[54] Ullman D.: *Elements of ML Programming*. Prentice Hall, 1998

[55] Yovine S.: *Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2)*. International Journal on Software Tools for Technology Transfer, vol. 1, 1997, 123–133