Maszyna stanów skończonych dla programisty systemów wbudowanych

Maszyna stanów skończonych jest pojęciem abstrakcyjnym i definiuje zachowanie systemów dynamicznych jako maszyny o skończonej liczbie stanów i skończonej liczbie przejść pomiędzy tymi stanami. Definicja ta może wydawać się dość abstrakcyjna dla typowego programisty, jednak jak się przekonamy, maszyny stanów skończonych odgrywają bardzo ważną rolę w programowaniu mikroprocesorów.

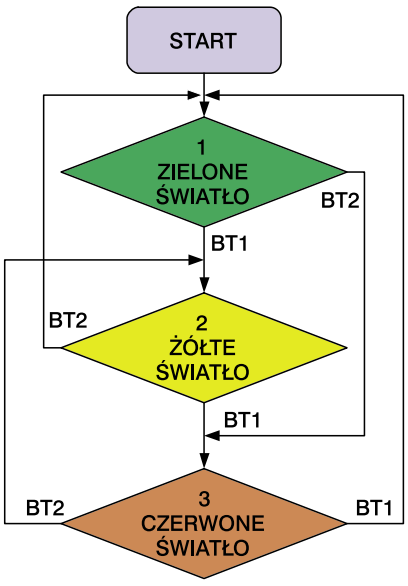
Nie skłamię, mówiąc, że większość programistów nieświadomie i bardzo często używa tych maszyn stanów. Aby zachęcić Czytelnika do dalszej lektury, powiem tylko, że ich używanie prowadzi do powstawania niezawodnych programów, których poprawność można udowodnić matematycznie. Z tego powodu artykuł jest lekturą obowiązkową dla wszystkich tych, dla których niezawodność programu jest kluczowa. Poniżej skupimy się na praktycznym wykorzystaniu maszyny stanów przy minimalnej dawce niezbędnej teorii.

Jeden przykład wart więcej niż 1000 słów

Zacznijmy nasze rozważania od prostego przykładu. Na **rys. 1** umieszczono schemat urządzenia do zmiany świateł. Zastosowano w nim dwa przyciski, które powodują zmianę świateł na „w przód" oraz „w tył". Na **rys. 2** pokazano algorytm, który powinien wykonywać program. Wynika z niego, że wciskając wielokrotnie przycisk BT1, powinno się uzyskać sekwencję przełączania świateł: *zielone, żółte, czerwone, zielone, żółte, czerwone...* Przycisk BT2 służy do zmiany sekwencji na „w tył", więc wciskając go, powinno się uzyskać sekwencję: *czerwone, żółte, zielone, czerwone, żółte...*

Napisanie programu dla tego algorytmu raczej nie sprawi nikomu problemu. Zanim Czytelnik zapozna się ze źródłem mojego programu (**list. 1**), proponuję napisać na kartce zarys swojej implementacji przedstawionego problemu. Wykonanie tego ćwiczenia pozwoli porównać nasze rozwiązania. Efekt takiego porównania może być bardzo ciekawy i pozwoli spojrzeć na problem z innej perspektywy. W swojej implementacji pominąłem wykonanie funkcji *pobierz_przycisk()* oraz *swiatlo()*, ponieważ są one zależne od użytego procesora, a my skupiamy się na istocie problemu, a nie na konkretnej implementacji. W naszych rozważaniach użyty sprzęt nie ma żadnego znaczenia, ponieważ można je snuć w odniesieniu do każdego mikroprocesora, również w dużych komputerach.

Analizując kod źródłowy z list. 1, można zauważyć, że użyłem zmiennej o nazwie *stan* do zapisu informacji o bieżącym, zaświeconym świetle. Taka organizacja kodu



Rys. 1.

Dodatkowe informacje:
Bibliotekę sm-lib można pobrać bezpłatnie ze strony internetowej *http://toan.pl*

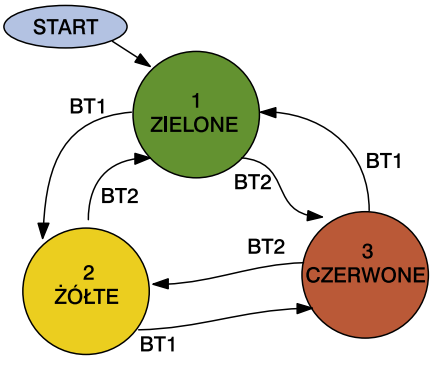
jest niczym innym, jak maszyną stanów. W programie mamy trzy możliwe stany, które odpowiadają zaświeconemu światłu: ZIELONE, CZERWONE oraz ŻÓŁTE. Porównajmy teraz kod programu z algorytmem.

Chociaż wykonaliśmy implementację algorytmu, to kod programu nie stanowi dokładnego jego opisu. Przydałby się jakiś sposób na implementację algorytmu bezpośrednio z diagramu. Ma rację ten, kto podejrzewa, że takie narzędzie zostanie za chwilę przedstawione. Zanim to jednak nastąpi, musimy przyjrzeć się, jak wygodnie reprezentować algorytmy w pamięci procesora.

Modelowanie algorytmu

Zastanówmy się, jak zapisać nasz algorytm do sterowania światłami. Najpierw wyróżnimy trzy stany:
STAN 1: ZIELONE
STAN 2: ŻÓŁTE
STAN 3: CZERWONE

Następnie stwórzmy warunki przejścia do następnego stanu:
STAN 1: ZIELONE
WARUNEK 1: PRZYCISK BT1 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 2 ŻÓŁTE
WARUNEK 2: PRZYCISK BT2 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 3 CZERWONE



Rys. 2.

List. 1.
```
#define ZIELONE  1
#define ZOLTE    2
#define CZERWONE 3

#define BT1      1
#define BT2      2

void swiatlo(int s)
{
...
}

char pobierz_przycisk()
{
...
}

int main()
{
  int stan=ZIELONE; // poczatkowy stan
  char przycisk;
  while(1) {
    przycisk=pobierz_przycisk();
    if (przycisk==BT1){
      switch(stan){
        case ZIELONE: stan=ZOLTE; break;
        case ZOLTE: stan=CZERWONE; break;
        case CZERWONE: stan=ZIELONE; break;
      }
    }
    if (przycisk==BT2){
      switch(stan){
        case ZIELONE: stan=CZERWONE; break;
        case ZOLTE: stan=ZIELONE; break;
        case CZERWONE: stan=ZOLTE; break;
      }
    }
    swiatlo(stan);
  }
  return 0;
}
```

STAN 2: ŻÓŁTE
WARUNEK 1: PRZYCISK BT1 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 3 CZERWONE
WARUNEK 2: PRZYCISK BT2 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 1 ZIELONE
STAN 3: CZERWONE
WARUNEK 1: PRZYCISK BT1 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 1 ZIELONE
WARUNEK 2: PRZYCISK BT2 WCIŚNIĘTY –> PRZEJŚCIE DO STANU 2 ŻÓŁTE

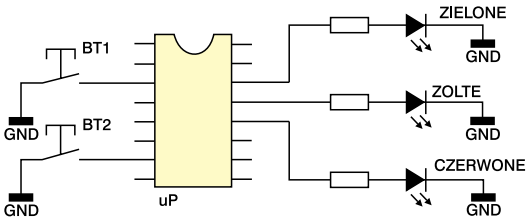
Udało nam się zapisać algorytm w formie słownika stanów. Zapiszmy słownik stanów w bardziej formalnej formie:

Taki zapis jest już wystarczający, żeby zaimplementować go właśnie w programie.

Stan	Warunki
1	BT1->2, BT2->3
2	BT1->3, BT2->1
3	BT1->1, BT2->2

Słownik:
Automat skończony: równoważne określenie dla maszyny stanów skończonych.
Graf: zbiór wierzchołków oraz połączeń między tymi wierzchołkami. Grafy mają duże praktyczne znaczenie dla informatyki i są uogólnieniem wielu struktur danych, np. drzew binarnych. Są stosowane np. w systemach GPS, ponieważ pozwalają łatwo rozwiązać problem komiwojażera.
UML: obiektowy język modelowania programów. Umożliwia m.in. modelowanie maszyny stanów, które będą implementowane w postaci obiektów np. w języku C++ lub Java.
Preprocesor języka C: program interpretujący, który przetwarza wstępnie kod języka C. Wszystkie instrukcje, które w programie zaczynają się od znaku #, są instrukcjami preprocesora. Najbardziej znane instrukcje to #include oraz #define.
Sieci Petriego: są uogólnieniem maszyny stanów i umożliwiają modelowanie współbieżnych zdarzeń.

Zanim jednak przedstawię bibliotekę oraz sposób implementacji takiego opisu w kodzie programu, przejdźmy o krok dalej i zastanówmy się, czy algorytm da się przedstawić jako zwykłą tabelę?



Rys. 3.

Macierzowa reprezentacja maszyny stanów

Przedstawienie algorytmu w formie macierzy jest jak najbardziej możliwe. Każdy, kto kiedykolwiek przerabiał teorie grafów, nie będzie miał problemów ze zrozumieniem macierzowej formy algorytmu. Algorytm jest w istocie grafem skierowanym i można utworzyć dla niego macierz przejść. Żeby to udowodnić, na **rys. 3** przedstawiłem algorytm w formie grafu skierowanego. Wierzchołek grafu odpowiada stanowi maszyny, natomiast łuk odpowiada warunkowi przejścia z jednego stanu do drugiego. W **tab. 1** pokazano macierz przejść dla naszego algorytmu. Wiersze reprezentują stan, z którego ma nastąpić przejście, natomiast kolumny stan docelowy, czyli stan, do którego nastąpi przejście ze stanu pierwotnego. Dane w tabeli odpowiadają warunkom, jakie powinny zostać spełnione aby przejście z jednego stanu do drugiego było możliwe. Macierz jest kwadratowa, czyli jej rozmiar to

Tab. 1.

	1	2	3
1		BT1	BT2
2	BT2		BT1
3	BT1	BT2	

N×N, gdzie N oznacza liczbę wszystkich stanów/wierzchołków.

Macierz stanów różni się od macierzy sąsiedztwa grafu tym, że ta pierwsza niesie informacje o warunkach przejścia, natomiast macierz sąsiedztwa zawiera informacje o liczbie dróg łączących wierzchołki. W przypadku, gdy chcemy przejść ze stanu 1 ZIELONE do stanu 3 CZERWONE, wybieramy w tabeli wiersz numer 1 oraz kolumnę numer 3 i sprawdzamy, jaki warunek kryje się w tabeli dla tego wyboru: BT2. Dysponując utworzoną macierzą, możemy sprawdzić, czy jest ona poprawna. Pierwszym testem jest sprawdzenie, czy w danym wierszu nie powtarzają się dwa takie same

List. 2.
```
#include „sm-lib/sm_seq.h”

#define ZIELONE  1
#define ZOLTE    2
#define CZERWONE 3

#define BT1      1
#define BT2      2

void swiatlo(int s)
{
...
}

char pobierz_przycisk()
{
...
}

void sygnalizacja()
{
  int k;
  k = pobierz_przycisk();

  SMS_BEGIN(SWIATLA,ZIELONE);
    SMS_STATE_BEGIN(SWIATLA, ZIELONE, swiatlo(ZIELONE) );
      SMS_STATE_COND(SWIATLA, k==BT1, ZOLTE);
      SMS_STATE_COND(SWIATLA, k==BT2, CZERWONE);
    SMS_STATE_END();
    SMS_STATE_BEGIN(SWIATLA, ZOLTE, swiatlo(ZOLTE) );
      SMS_STATE_COND(SWIATLA, k==BT1, CZERWONE);
      SMS_STATE_COND(SWIATLA, k==BT2, ZIELONE);
    SMS_STATE_END();
    SMS_STATE_BEGIN(SWIATLA, CZERWONE, swiatlo(CZERWONE) );
      SMS_STATE_COND(SWIATLA, k==BT1, ZIELONE);
      SMS_STATE_COND(SWIATLA, k==BT2, ZOLTE);
    SMS_STATE_END();
  SMS_END(SWIATLA,ZIELONE);
}

int main()
{
  while(1) {
    sygnalizacja();
  }
  return 0;
}
```

List. 3.
```
void sygnalizacja()
{
  int k;
  k = pobierz_przycisk();
  static unsigned char current_state = 1;
  switch ( current_state ) {
    case 1: {
      swiatlo(1);
      if( k==1 ) { current_state = 2; break; };
      if( k==2 ) { current_state = 3; break; };
      break; };
    case 2: {
      swiatlo(2);
      if( k==1 ) { current_state = 3; break; };
      if( k==2 ) { current_state = 1; break; };
      break; };
    case 3: {
      swiatlo(3);
      if( k==1 ) { current_state = 1; break; };
      if( k==2 ) { current_state = 2; break; };
      break; };
    default:
      current_state = 1;
  }
}

int main()
{
  while(1) {
    sygnalizacja();
  }
  return 0;
}
```

warunki. Taka sytuacja nie może mieć miejsca, ponieważ przy spełnionym warunku powstałaby niejednoznaczność, do którego stanu należy przejść. Kolejnym testem może być sprawdzenie przekątnej macierzy. Na przekątnej nie powinno być warunków, ponieważ nie ma sensu zmiana stanu na ten sam stan, dlatego że jest to marnowanie mocy procesora.

Biblioteka dla języka C

Przedstawię poniżej bardzo prostą bibliotekę napisaną w preprocesorze języka C. Użyłem makra preprocesora z tego względu, że biblioteka jest dedykowana dla małych mikroprocesorów, więc rozmiar kodu i wydajność ma kluczowe znaczenie. Biblioteka potrzebuje dla definicji każdej maszyny stanów tylko 1 bajta pamięci. Jest wieloplatformowa i bez żadnych zmian można jej używać na każdym mikroprocesorze. Jedynym ograniczeniem jest kompilator, który powinien obsługiwać makra preprocesora. Na całą bibliotekę składają się tylko dwa pliki: *sm_cond.h* oraz *sm_table.h*. Pierwszy plik zawiera implementację maszyny stanów w formie słownika stanów. Natomiast drugi plik zawiera macierzową implementację maszyny stanów. Biblioteka jest sprawdzona w boju, ponieważ użyłem jej już w kilku programach dla różnych urządzeń. Zanim jednak pokażę przykład użycia biblioteki, odpowiemy sobie na pytanie: Po co stosować tę bibliotekę, gdy można samemu zaprogramować tę trywialną funkcjonalność? Otóż zastosowanie biblioteki ma duże znaczenie dla stabilności rozwiązania. Biblioteka jest dobrze przetestowana i dzięki temu nie potrzebujemy testować funkcjonalności maszyny stanów, a jedynie sam algorytm. Zapewne wielu Czytelników zechce zastosować tę bibliotekę w swoim programie, więc wyjdą na jaw błędy, których sam nie wykryłem, co sprawi, że biblioteka będzie jeszcze bardziej niezawodna. Spójrzmy teraz na **list. 2**. Zawiera on odpowiednik programu z list. 1, ale napisany z użyciem biblioteki *sm_cond.h*. Moglibyście spytać, gdzie jest właściwy program? Odpowiedź jest prosta: Program jest zawarty w regułach maszyny stanów. Prze-

List. 4.
```
#include <stdio.h>
#include „sm-lib/sm_seq.h”

#define ON     1
#define OFF    0

#define TRUE   1
#define FALSE  0

#define GETC()   getchar()
#define PUTS(X)  printf(X)

static int echo;

void init()
{
  PUTS(„INIT OK\n”);
  echo=OFF;
}

void at()
{
  PUTS(„OK\n”);
}

void ati()
{
  PUTS(„MODEM FIRMWARE v.1.0.0\n”);
}

void error()
{
  PUTS(„BLAD\n”);
}

void ate()
{
  echo=!echo;
  if (echo==ON)
    printf(„ECHO ON\n”);
  else
    printf(„ECHO OFF\n”);
}

char pobierz_znak()
{
  int k = GETC();
  if ( (echo==ON) ) printf(„%c”,k);
  return k;
}

void modem()
{
  static char c=0;

  SMS_BEGIN(MODEM,1);
    SMS_STATE_BEGIN(MODEM, 1, init() );
      SMS_STATE_COND(MODEM, TRUE, 2);
    SMS_STATE_END();
```

List. 4. c.d.
```
    SMS_STATE_BEGIN(MODEM, 2, c=pobierz_znak() );
      SMS_STATE_COND(MODEM, c==’a’,  3);
      SMS_STATE_COND(MODEM, c==’ ,,  2);
      SMS_STATE_COND(MODEM, c==’\n’, 2);
      SMS_STATE_COND(MODEM, c!=0,   10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 3, c=pobierz_znak() );
      SMS_STATE_COND(MODEM, c==’t’, 4);
      SMS_STATE_COND(MODEM, c!=0,   10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 4, c=pobierz_znak() );
      SMS_STATE_COND(MODEM, c==’i’, 5);
      SMS_STATE_COND(MODEM, c==’e’, 6);
      SMS_STATE_COND(MODEM, c==’z’, 7);
      SMS_STATE_COND(MODEM, c==’ ,, 4);
      SMS_STATE_COND(MODEM, c==’\n’,13);
      SMS_STATE_COND(MODEM, c!=0,   10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 5, c=pobierz_znak() );
      SMS_STATE_COND(MODEM, c==’\n’, 8);
      SMS_STATE_COND(MODEM, c==’ ,,  11);
      SMS_STATE_COND(MODEM, (c!=0),  10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 6, c=pobierz_znak() );
      SMS_STATE_COND(MODEM, c==’\n’, 9);
      SMS_STATE_COND(MODEM, c==’ ,,  12);
      SMS_STATE_COND(MODEM, (c!=0),  10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 7, c=pobierz_znak() ); //ATZ
      SMS_STATE_COND(MODEM, c==’\n’, 1);
      SMS_STATE_COND(MODEM, (c!=0),  10);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 8, ati() ); //ATI
      SMS_STATE_COND(MODEM, TRUE, 2);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 9, ate(); ); //ATE
      SMS_STATE_COND(MODEM, TRUE, 2);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 10, error() ); // ERROR
      SMS_STATE_COND(MODEM, TRUE, 2);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 11, ati() ); //ATI
      SMS_STATE_COND(MODEM, TRUE, 4);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 12, ate() ); //ATE
      SMS_STATE_COND(MODEM, TRUE, 4);
    SMS_STATE_END();
    SMS_STATE_BEGIN(MODEM, 13, at() );  //AT
      SMS_STATE_COND(MODEM, TRUE, 2);
    SMS_STATE_END();
  SMS_END(MODEM,1);
}
int main()

{
  while(1) {
    modem();
  }
  return 0;
```

śledźmy teraz instrukcje odpowiedzialne za definicję maszyny stanów.

Makro SM_BEGIN(SWIATLA, ZIELONE) definiuje nową maszynę stanów o nazwie SWIATLA. Nazwa SWIATLA musi być unikalna. Nazwanie maszyny stanów jest konieczne z tego względu, że możemy zdefiniować wiele maszyn stanów w swoim programie i mógłby pojawić się konflikt nazw. Stała ZIELONE w wywołaniu tego makra oznacza startowy stan maszyny. Stałe ZIELONE, CZERWONE i ZOLTE muszą być unikalnymi liczbami i służą do oznaczenia stanu maszyny. Następnie w kodzie znajdują się trzy definicje pary:

STAN<–>LISTA WARUNKÓW.

Makro SM_STATE_BEGIN(SWIATLA, ZIELONE, swiatlo(ZIELONE)) rozpoczyna parę i zawiera trzy argumenty. Pierwszy argument to nazwa maszyny, drugi argument to stan, którego dotyczy para, natomiast trzeci argument to akcja do wykonania dla tego stanu. Akcją może być wywołanie funkcji tak jak w naszym przykładzie lub można umieścić bezpośrednio jakieś instrukcje w tym miejscu. Makro SM_STATE_COND(SWIATLA, k==BT1, ZOLTE) definiuje warunek dla danego stanu i zawiera również trzy argumenty. Pierwszy argument to nazwa maszyny, drugi argument to warunek, który powinien zostać spełniony, żeby przejść z tego stanu do stanu podanego w argumencie trzecim. Możemy oczywiście zdefiniować sobie dowolną liczbę warunków oraz stanów. Musimy jednak uważać, aby całość był spójna i żeby nigdy nie było sytuacji zdefiniowania warunku przejścia do stanu, którego definicja nie istnieje. Jeśli chcemy zobaczyć czysty kod języka C bez preprocesora, to możemy użyć opcji -E kompilatora GCC (inne kompilatory także powinny posiadać taką opcję). Na **list. 3** pokazano kod po przetworzeniu go przez preprocesor. Jak widać na listingu, biblioteka nie tworzy żadnych wywołań do swoich wewnętrznych funkcji, tylko generuje kod i wstawia go w miejscu wywołania makr. Takie podejście zapewnia dużą wydajność rozwiązania i łatwą analizę takiego kodu.

Praktyczny przykład: analizator komend AT

Komendy AT są szeroko stosowane w modemach. Wybór takiego przykładu jest nieprzypadkowy. Bardzo często w praktyce elektronika trzeba utworzyć protokół transmisji danych do komunikacji z danym urządzeniem. Ten przykład pokaże, w jaki sposób użyć do tego maszyny stanów. W prezentowanym przykładzie została zrobiona analiza 4 podstawowych komend AT:

at – zwraca OK
ati – informacje o modemie
ate – włącza/wyłącza echo
atz – reset urządzenia

Komendy można łączyć w jednym ciągu np. dla ciągu znaków „ati e z” zostaną wykonane komendy: ati, ate oraz atz. Nasz analizator musi być odporny na błędy oraz na różne sytuacja np. „ati e z”, czyli wiele spacji rozdzielających w ciągu.

Kod programu znajduje się na **list. 4**. Kod w obecnej postaci można skompilować dowolnym kompilatorem na komputer PC. Jeśli będziemy chcieli skompilować go na mikroprocesor i wykorzystać UART do jego obsługi, to musimy jedynie zmienić definicje GETC i PUTS oraz dodać inicjalizację odpowiednich urządzeń specyficzną dla danego mikroprocesora. Implementując analizator komend w oparciu o zaprezentowaną bibliotekę, oszczędzamy pamięć RAM kosztem pamięci programu (np. Flash). Oszczędność pamięci RAM jest dużą zaletą, gdy piszemy program na mały mikroprocesor, ponieważ zazwyczaj mamy do dyspozycji niewiele pamięci RAM, a za to dużo pamięci programu typu Flash.

Podsumowanie

Może się wydawać, że przedstawiona biblioteka jest wręcz idealna do każdego zastosowania, ale niestety tak nie jest. Nie ma prostej odpowiedzi na pytanie, kiedy należy ją stosować. Są jedynie pewne przesłanki, które pomagają to ustalić.

Jeśli piszemy program i chcemy użyć zmiennej statycznej w danej funkcji, czyli chcemy pamiętać stan tej zmiennej w kolejnych krokach, to jest to dla nas sygnał, że należy się zastanowić nad użyciem maszyny stanów. Podobnie ma się sprawa z rekurencją. Często lepszym rozwiązaniem od rekurencji będzie użycie maszyny stanów. Jeśli np. chcemy zbudować wielopoziomowe menu w swoim urządzeniu to wręcz musimy zastosować maszynę stanów. Przestrzegam także przed tworzeniem maszyn o bardzo dużej liczbie możliwych stanów. Dużo lepszym pomysłem jest podzielenie całego programu na logiczne części i zastosowania kilku maszyn stanów zamiast jednej bardzo rozbudowanej. Przy tworzeniu algorytmu sterującego bardzo pomocne mogą okazać się programy komputerowe do tworzenia diagramów algorytmu. Do tworzenia takich diagramów wystarczy program OpenOffice. Jeśli chcielibyśmy dedykowane narzędzie to można użyć programów do modelowania sieci Petriego. Zachęcam wszystkich Czytelników do własnych prób w tej dziedzinie. Polecam zainteresować się także sieciami Petriego z tego względu, że są uogólnieniem maszyny stanów. Będę również wdzięczny za wszelkie uwagi oraz sugestie dotyczące artykułu oraz prezentowanej biblioteki, które można wysłać na mój e-mail.

Tomasz Orłowski
tomek@toan.pl