Bezpieczeństwo w inżynierii oprogramowania

Metody formalne

Arkadiusz Chrobot

Katedra Systemów Informatycznych

7 grudnia 2024

Plan

Wstęp

Metody formalne są, obok testowania statycznego i dynamicznego, są trzecią możliwością kontroli i zapewniania jakości oprogramowania [4]. Bazują one silnie na dziedzinach matematyki, takich jak logika i teoria zbiorów, stąd w praktyce ich stosowanie ogranicza się głównie do rozwoju i utrzymania oprogramowania o zastosowaniach krytycznych. W innych przypadkach użycie metod formalnych może być zbyt kosztowne.

Ten wykład ma charakter wprowadzający i stanowi przegląd wybranych metod formalnych. Dokładne opisanie nawet jednej z nich wymagałoby przedstawienia aparatu matematycznego, na którym jest ona oparta i zajęłoby kilka wykładów.

Wstęp

Poziomy zastosowania metod formalnych

Metody formalne są stosowane w kilku fazach tworzenia oprogramowania, takich jak *specyfikacja*, *projektowanie*, *implementacja* i *weryfikacja* [1]. W każdej z tych czynności mogą one być stosowane w różnym stopniu:

- Wyłącznie formalna specyfikacja (pozostałe fazy bez użycia metod formalnych).
- Formalna specyfikacja, dopracowywanie (ang. *refinement*) i weryfikacja (częściowe dowodzenie).
- Formalna specyfikacja, dopracowanie i weryfikacja (z obszernym stosowanie dowodzenia).

Te poziomy zastosowania metod formalnych są wymagane przy ubieganiu się o najwyższe stopnie certyfikacji ▸ Common Criteria.

Wstęp

Implementacja w metodach formalnych

W metodach formalnych implementacja jest *wyprowadzana* (ang. *derived*) [1] z formalnej specyfikacji (S) przez *stopniowe dopracowywanie* (ang. *stepwise refinement*), czyli otrzymywanie coraz konkretniejszego modelu (M) oprogramowania, którego ostateczna formą jest implementacją (E): $S = M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq M_3 \ldots \sqsubseteq M_n = E$

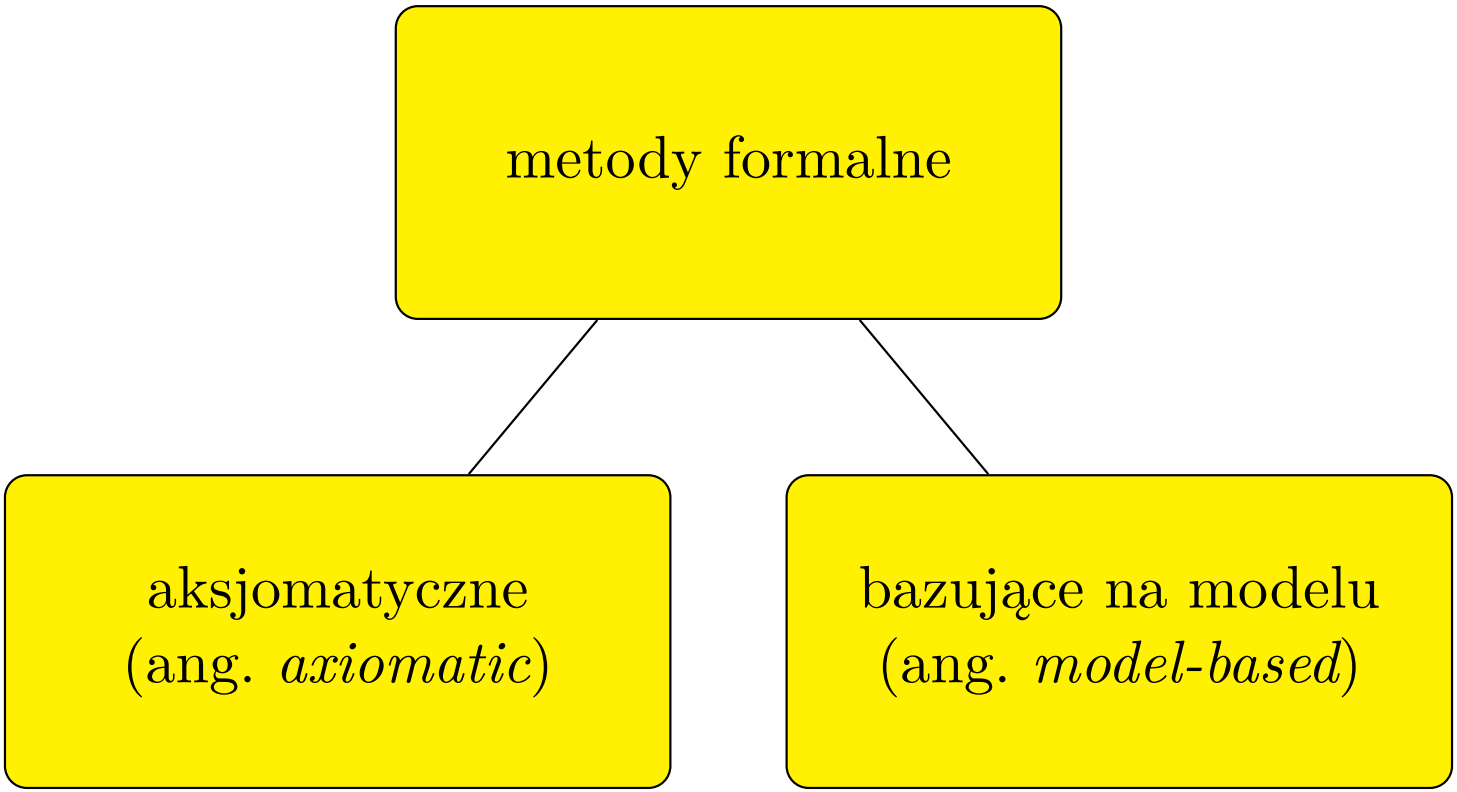
Wstęp

Rodzaje metod formalnych

Opracowano wiele metod formalnych, ale można je wszystkie podzielić na dwie grupy (Rysunek 1): metody aksjomatyczne i bazujące na modelach [4, 1]. Historycznie pierwsze pojawiły się te zaliczane do kategorii aksjomatycznych, dlatego zostaną omówione jako pierwsze.

Wstęp

Rodzaje metod formalnych



Klasyfikacja metod formalnych (na podstawie [4])

Metody aksjomatyczne

Celem metod aksjomatycznych jest zapewnienie, że tworzone oprogramowanie będzie posiadało określone, pożądane własności. Aby go osiągnąć opisują one te atrybuty przy pomocy zdań logicznych, których poprawność jest następnie dowodzona, np. przy pomocy indukcji matematycznej. Zaletą tego podejścia jest to, że nie wymusza ono konkretnej implementacji oprogramowania. Programiści mogą zastosować dowolną, pod warunkiem, że będzie ona spełniała wskazane własności. Z drugiej strony ta zalega może się okazać także wadą, bo nie ma bezpośredniej zależności między tymi atrybutami, a implementacją. Dlatego dodatkowym warunkiem w przypadku stosowania metod aksjomatycznych jest upewnienie się, że proponowane własności są w praktyce osiągalne.

Niezmienniki i logika Hoare'a

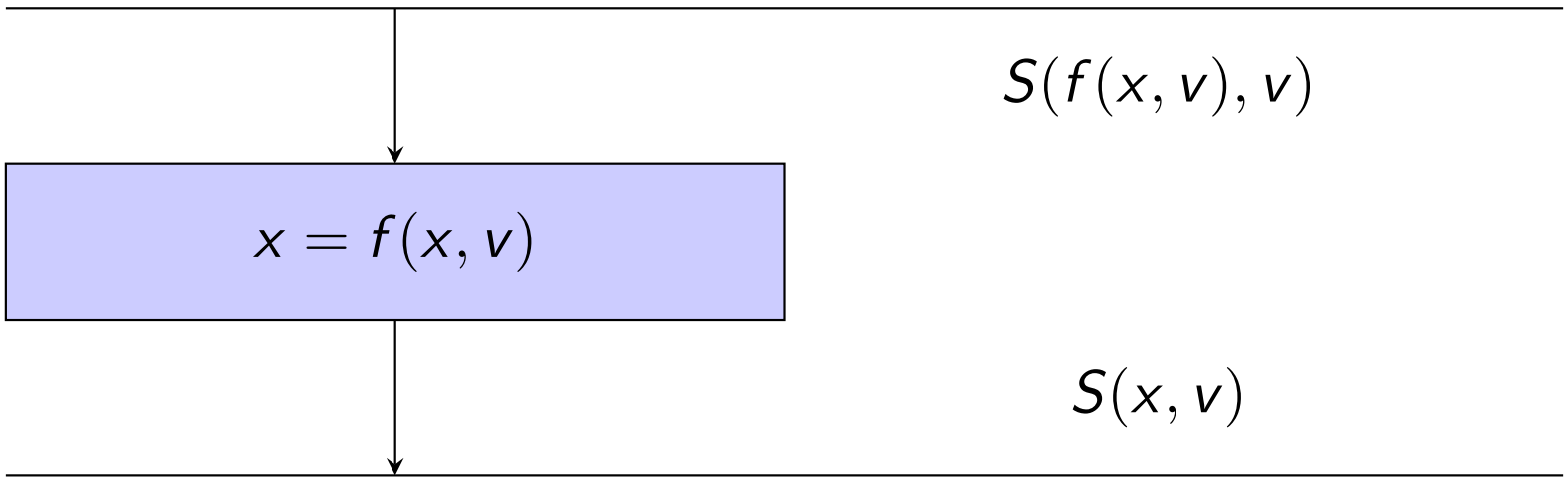
Pionierskie prace w zakresie metod formalnych wykonał w latach 60. ubiegłego wieku amerykański informatyk Robert Floyd. Wykazał on, że można dowodzić poprawności oprogramowania na drodze matematycznej i zaproponował stosowanie *niezmienników*, inaczej zwanych *asercjami*, czyli własności opisanych za pomocą wyrażeń logicznych, które *powinny* być spełnione przez *stan* programu. Poprzez stan należy tu rozumieć wartości zmiennych używanych w objętym asercją fragmencie kodu. W oryginalnej metodzie Floyda asercje są związane z każdą krawędzią schematu blokowego, reprezentującego program. Asercja znajdująca się przed blokiem reprezentującym operację jest nazywana *wejściową* (ang. *entry*), a ta za nim *wyjściową* (ang. *exit*). Jeśli można wykazać, że asercja wyjściowa dla danej operacji jest konsekwencją asercji wyjściowej to można dowieść, że niezmiennik na końcu programu będzie prawdziwy, jeśli ten na początku także taki jest.

Niezmienniki i logika Hoare'a

Asercje mogą obejmować kilka różnych konstrukcji w kodzie programu [2]:

- *Instrukcje sekwencyjne* — w tym przypadku asercje mogą być umieszczone przed i po każdej takiej instrukcji lub bloku instrukcji. Przykładem jest instrukcja $x = f(x, v)$ przypisania z Rysunku 2. Litera S oznacza niezmiennik, x zmienną, v wektor składający się ze wszystkich zmiennych w programie, a $f(x, v)$ wyrażenie określone na x i zmiennych z wektora v. Jeśli asercja $S(f(x, v), v)$ jest prawdziwa przed wykonaniem instrukcji przypisania, niezmiennik $S(x, v)$ jest prawdziwy po jej wykonaniu.

Niezmienniki i logika Hoare'a

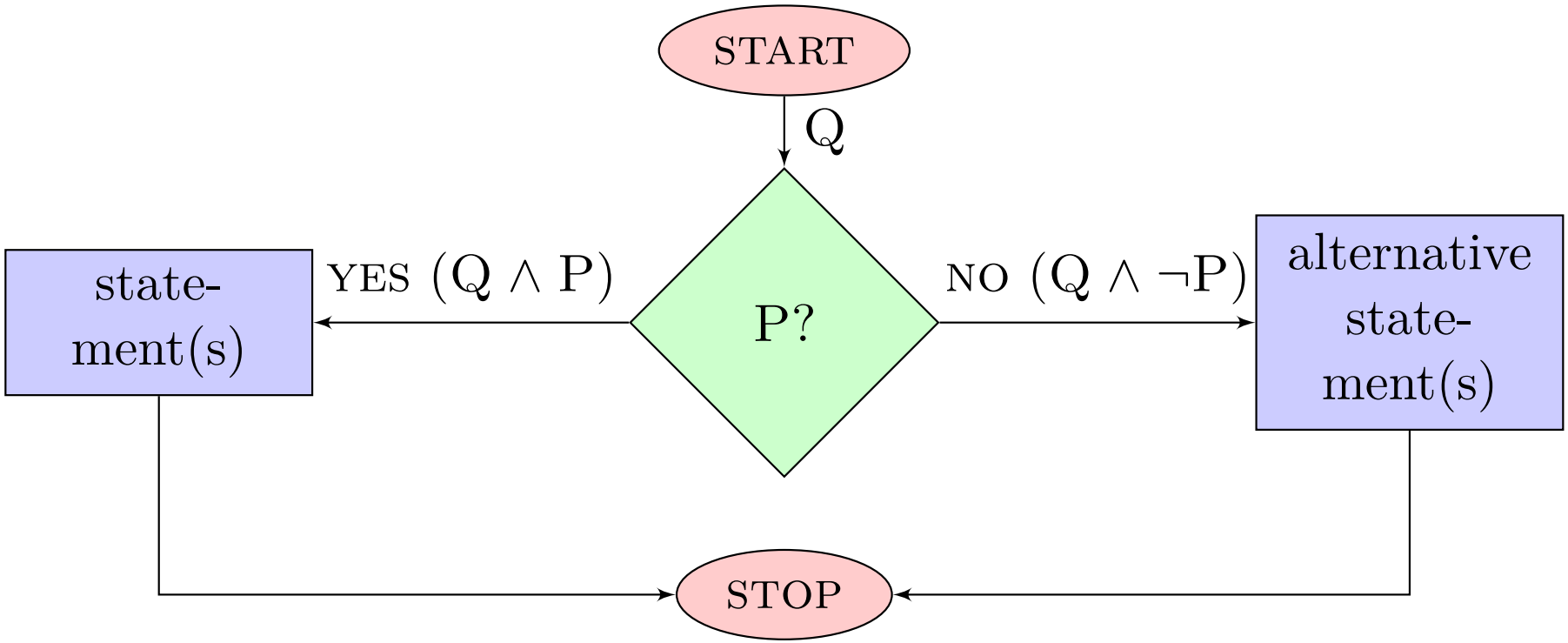


Asercja dla instrukcji przypisania w schemacie blokowym (na podstawie: [1])

Niezmienniki i logika Hoare'a

- *Instrukcje wyboru* — do tych instrukcji zalicza się takie, jak `if` i `case`. Podczas wykonania programu dokonywany jest wybór jednej z możliwości. Aby dowieść poprawności działania takiej instrukcji należy zweryfikować każdą możliwość z osobna. Przykładem jest instrukcja `if`, której schemat blokowy przedstawia Rysunek 3. Jeśli asercja Q, jest prawdziwa i spełniony jest warunek P, to prawdziwy będzie niezmiennik $Q \wedge P$. Jeśli zaś warunek P nie będzie fałszywy, to spełniona będzie asercja $Q \wedge \neg P$.

Niezmienniki i logika Hoare'a



Asercje dla rozgałęzień instrukcji warunkowej w schemacie blokowym (na podstawie: [1])

Niezmienniki i logika Hoare'a

- *Iteracje* inaczej *pętle* — poprawność pętli jest dowodzona w trzech etapach. Na początku należy wykazać, że asercja wejściowa staje się prawdziwa po zainicjowaniu pętli. Następnie należy udowodnić, przy pomocy indukcji matematycznej, że jest ona prawdziwa po każdej iteracji. Następnie należy przeprowadzić dowód, że pętla się zakończy i da poprawny wynik, czyli że będzie spełniona asercja wyjściowa.
- *Podprogramy* — przez podprogramy należy rozumieć *funkcje*, *procedury* i/lub *metody*. Ich poprawności dowodzi się umieszczając asercję wejściową i wyjściową w miejscu ich wywołania. W przeciwieństwie jednak do „zwykłych" instrukcji mają one raczej formę kontraktu, niż stwierdzenia faktu. Dopiero dowodząc poprawności *treści*, czyli *definicji* podprogramu można zapewnić, że niezmiennik wyjściowy będzie spełniony, jeśli prawdziwy będzie także wejściowy.

Niezmienniki i logika Hoare'a

Dzięki pracom Floyda wprowadzono do języków programowania makra (języki C/C++) lub słowa kluczowe (Java, Python) `assert`, pozwalające sprawdzać niezmienniki w trakcie działania programu. Dodatkowo prace te stały się inspiracją dla brytyjskiego informatyka sir Tony'ego Hoare'a do opracowania specjalnej logiki pozwalającej wnioskować o poprawności programu, nazwanej jego nazwiskiem [1]. Podstawowym zapisem w tej logice jest $P\{a\}Q$, gdzie P jest warunkiem wstępnym (ang. *precondition*), Q warunkiem końcowym (ang. *postcondition*), a a jest fragmentem kodu. Oba warunki są jednocześnie predykatami i asercjami, podobnie jak cały zapis, który wyraża *częściową poprawność* (ang. *partial correctness*) fragmentu a. Ten rodzaj poprawności zachodzi, gdy Q jest spełniony po wykonaniu a w stanie, w którym P jest prawdziwy. *Całkowita poprawność* (ang. *total correctness*), zapisywana $\{P\}a\{Q\}$ wymaga, aby Q był prawdziwy po zakończeniu wykonania a w *dowolnym* stanie, w którym P jest spełniony.

Niezmienniki i logika Hoare'a

Logika Hoare'a oparta jest na częściowej poprawności i zawiera aksjomaty oraz reguły wnioskowania pozwalające opisać zależności między wejściowymi i wyjściowymi warunkami. Oto kilka przykładowych konstrukcji w tej logice [1]:

- instrukcja pusta (ang. *skip*): $P\{skip\}Q$,
- instrukcja przypisania: $P^x_e\{x := e\}P$,
- instrukcja złożona (ang. *compound*): $\frac{P\{S_1\}Q,\ Q\{S_2\}R}{P\{S_1;S_2\}R}$,
- instrukcja warunkowa: $\frac{P\wedge B\{S_1\}Q,\ P\wedge\neg B\{S_2\}Q}{P\{\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\}Q}$,
- pętla `while`: $\frac{P\wedge B\{S\}P}{P\{\textbf{while } B \textbf{ do } S\}P\wedge\neg B}$.

Język CSP

Profesor Hoare jest również autorem języka formalnego CSP (ang. *Communicating Sequential Processes*), który przeznaczony jest do opisywania schematów interakcji między procesami lub wątkami w systemach współbieżnych [5]. Ten język jest też przykładem *rachunku procesów* (ang. *process calculi*). Podstawowym pojęciem w takim języku jest *proces*, który wchodzi w interakcję ze swoim środowiskiem (innymi procesami oraz zasobami) [1]. Wyrażenie $(a \ ? \ P)$ oznacza proces, który najpierw jest zaangażowany w zdarzenie (ang. *event*) a, a następnie staje się procesem P. W CSP procesy komunikują się przy pomocy *kanałów*. Przykładowo $(c?.x \ P_x)$ opisuje proces, który najpierw otrzymuje wartość x kanałem c, a potem się zachowuje jak proces P_x. Z kolei nadawca komunikatu może być opisany następująco: $(c!eP)$, czyli jest to proces, który nadaje kanałem c wartość wyrażenia e, a potem zachowuje się jak proces P.

Język CSP

CSP pozwala także opisywać procesy niekończące się przy pomocy rekurencji oznaczanej jako $(\mu X) \cdot F(X)$. Przykładowo automat sprzedający czekoladę ($choc$) po wrzuceniu monety ($coin$) może być opisany następująco:

$$VMS = \mu X : \{coin, choc\} \cdot (coin?(choc?X))$$

Ten zapis oznacza, że proces X reprezentujący oprogramowanie automatu do sprzedaży czekolady (VMS), po otrzymaniu monety ($coin$) wydaje czekoladę ($choc$) i staje się na powrót procesem X.

Usprawnienia metod aksjomatycznych

Klasyczna logika nie jest wystarczająca do opisu funkcji częściowych (ang. *partial functions*) oraz zależności czasowych występujących w systemach współbieżnych. *Funkcja częściowa*, to funkcja matematyczna, która nie ma przypisanych wartości w przeciwdziedzinie dla każdego elementu określonego zbioru [1]. Jednym z najprostszych przykładów jest funkcja hiperboliczna $f(x) = \frac{1}{x}$. Jej dziedziną jest zbiór wszystkich liczb rzeczywistych z pominięciem zera. Problem polega na tym, że w oprogramowaniu to zero może się pojawić jako wartość wejściowa. David Parnas zaproponował rozszerzenie klasycznej logiki, w którym wyrażenie $\frac{1}{0}$ jest po prostu fałszywe. Pozostałe wartości funkcji hiperbolicznej są prawdziwe. Problem opisu systemów współbieżnych rozwiązano wprowadzając *logiki temporalne* (ang. *temporal logic*), w których występują operatory pozwalające wskazać, że wyrażenie ma wartość prawdziwą lub fałszywą obecnie lub będzie ją miało w przyszłości (chwilowo lub przez dłuższy czas) lub miało ją w przeszłości (krótko lub dłużej).

Metody bazujące na modelu

Zgodnie z nazwą metody formalne oparte na modelu pozwalają na tworzenie *abstrakcyjnego*, czyli zawierającego tylko niezbędne szczegóły, modelu oprogramowania. Ten model ma charakter matematyczny i opisuje *stan* programu oraz *operacje*, które mogą go zmieniać. Dzięki temu modelowi można nie tylko badać bieżące zachowanie oprogramowania, ale także wnioskować o jego przyszłym zachowaniu.

Specyfikacja Z

Specyfikacja Z, nazywana również *Notacją Z* jest jednym z przykładów metod formalnych opartych na modelach. Oficjalnie specyfikacja Z nazywa się *formalnym językiem specyfikacji Z*. Opracował ją francuski informatyk Jean-Raymond Abrial wraz ze Stephenem A. Schumanem i Bertrandem Meyerem. Większą część prac Abrial wykonał na Uniwersytecie w Oxfordzie, pod kierownictwem Tony'ego Hoare'a. W roku 2002 ta metoda został opisana w standardzie ISO. Podstawowym elementem tego języka jest *schemat*, który może opisywać *stan* lub *operacje* oprogramowania. Dodatkowo, specyfikacja Z zawiera również operatory, które pozwalają wykonywać działania na schematach. Notacja ta bazuje na dziedzinach matematyki dotyczących zbiorów, wielozbiorów (ang. *bags*), relacji, funkcji i sekwencji. Warto zaznaczyć, że zbiory w specyfikacji Z mają określone typy i obok zbiorów numerycznych, takich jak zbiór liczb naturalnych $\mathbb{N}$, całkowitych $\mathbb{Z}$ oraz rzeczywistych $\mathbb{R}$ można również definiować własne.

Specyfikacja Z

Przykład — specyfikacja systemu bibliotecznego

Schemat 4 przedstawia stan systemu bibliotecznego opisany notacją Z [1]. Zbiory *on-shelf*, *missing* i *borrowed* są podzbiorami zbioru wszystkich podzbiorów identyfikatorów książek ($\mathbb{P}$ Bkd-Id) i reprezentują, odpowiednio, książki, które są do wypożyczenia, zgubione i wypożyczone. Aby stan był spójny i poprawny wymaga się, aby iloczyny tych podzbiorów dawały w wyniku zbiór pusty.

Library

$$\text{on-shelf}, \text{missing}, \text{borrowed} : \mathbb{P}\,\text{Bkd-Id}$$

$$\text{on-shelf} \cap \text{missing} = \emptyset$$
$$\text{on-shelf} \cap \text{borrowed} = \emptyset$$
$$\text{borrowed} \cap \text{missing} = \emptyset$$

Specyfikacja systemu bibliotecznego w specyfikacji Z (źródło: [1])

Specyfikacja Z

Przykład — specyfikacja systemu bibliotecznego

Schemat 5 opisuje operację wypożyczenia książki. Jest to operacja zmieniająca stan biblioteki, stąd oznaczona jest symbolem Δ. Gdyby ten stan nie ulegał zmianie, to byłaby oznaczona symbolem Ξ. Zmienna b jest *zmienną wejściową*, więc jest oznaczona jako $b?$, gdyby była *zmienną wyjściową*, to byłaby oznaczona $b!$, a gdyby jej stan ulegał zmianie, to po modyfikacji byłaby oznaczona b'.

Borrow

ΔLibrary

$b? : \text{Bkd-Id}$

$b? \in \text{on-shelf}$

$\text{on-shelf} = \text{on-shelf} \setminus \{b?\}$

$\text{borrowed'} = \text{borrowed} \cup \{b?\}$

Specyfikacja operacji wypożyczenia książki w specyfikacji Z (źródło: [1])

Specyfikacja Z

Zmienna b w schemacie 5 reprezentuje książkę. Aby operacja jej wypożyczenia była poprawna, to warunek początkowy wymaga, aby należała ona do podzbioru książek, które nie są wypożyczone. W wyniku wykonania operacji ta książka powinna być usunięta z podzbioru książek niewypożyczonych, a dodana do zbioru książek wypożyczonych.

Specyfikacja Z definiuje również operacje na schematach, takie jak *włączanie* (ang. *inclusion*), *łączenie* (ang. *merge*) — przy pomocy operacji iloczynu logicznego, sumy logicznej, koniunkcji i równoważności — oraz operacja kompozycji (ang. *composition*), dla której został zdefiniowany odpowiedni algorytm. Te operatory służą ukonkretnianiu (ang. *reification*) stanu i dekompozycji operacji, aby przejść z opisu abstrakcyjnego, w którym używane są elementy niewystępujące w językach programowania, do implementacji. Każdemu takiemu działaniu towarzyszy dowodzenie poprawności, wykonane ręcznie lub przy pomocy narzędzi.

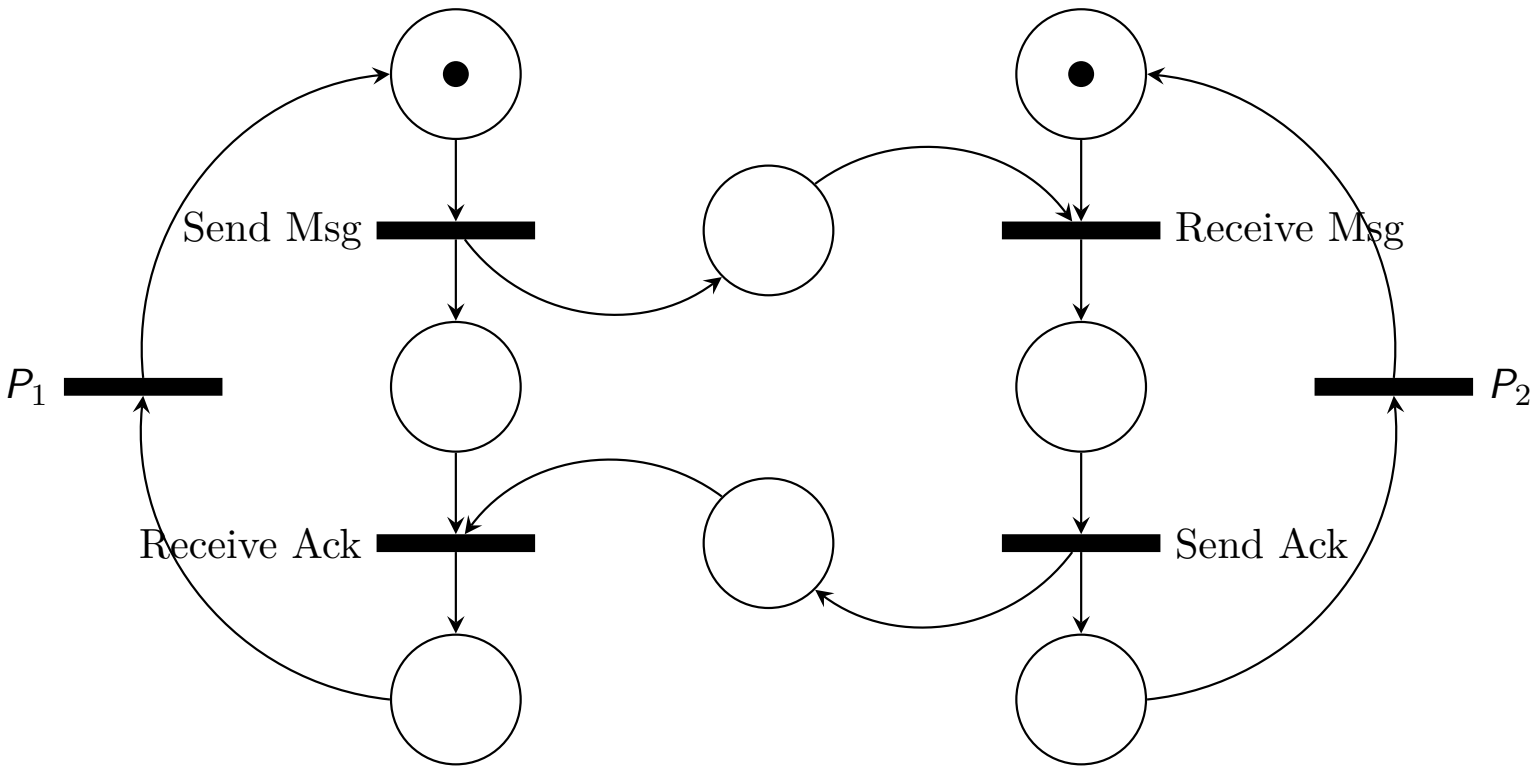
Sieci Petriego

Sieci Petriego są metodą opisu systemów współbieżnych, która została opracowana przez niemieckiego matematyka Adama Petriego i nazwana jego nazwiskiem [6, 7]. Taka sieć, to graf skierowany, w którym istnieją dwa rodzaje wierzchołków: *miejsca* (oznaczane graficznie okręgami) i *przejścia* (oznaczane graficznie prostokątami lub kwadratami). Przejścia reprezentują działania (ang. *action*), natomiast miejsca rozproszony stan. Każde miejsce może zawierać token (znacznik, żeton), który oznacza, że dany stan jest aktywny. W sieci mogą także występować miejsca, które mogą zawierać więcej tokenów. W takim wypadku krawędź dochodząca do takiego miejsca może mieć określoną wagę, wyrażającą pojemność tego miejsca. Miejsca i przejścia występują naprzemiennie w sieciach Peteriego. Aby token mógł przejść z jednego miejsca do drugiego (oprogramowanie mogło zmienić stan) musi zostać *odpalone* znajdujące się między nimi miejsce (wykonana operacja).

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.

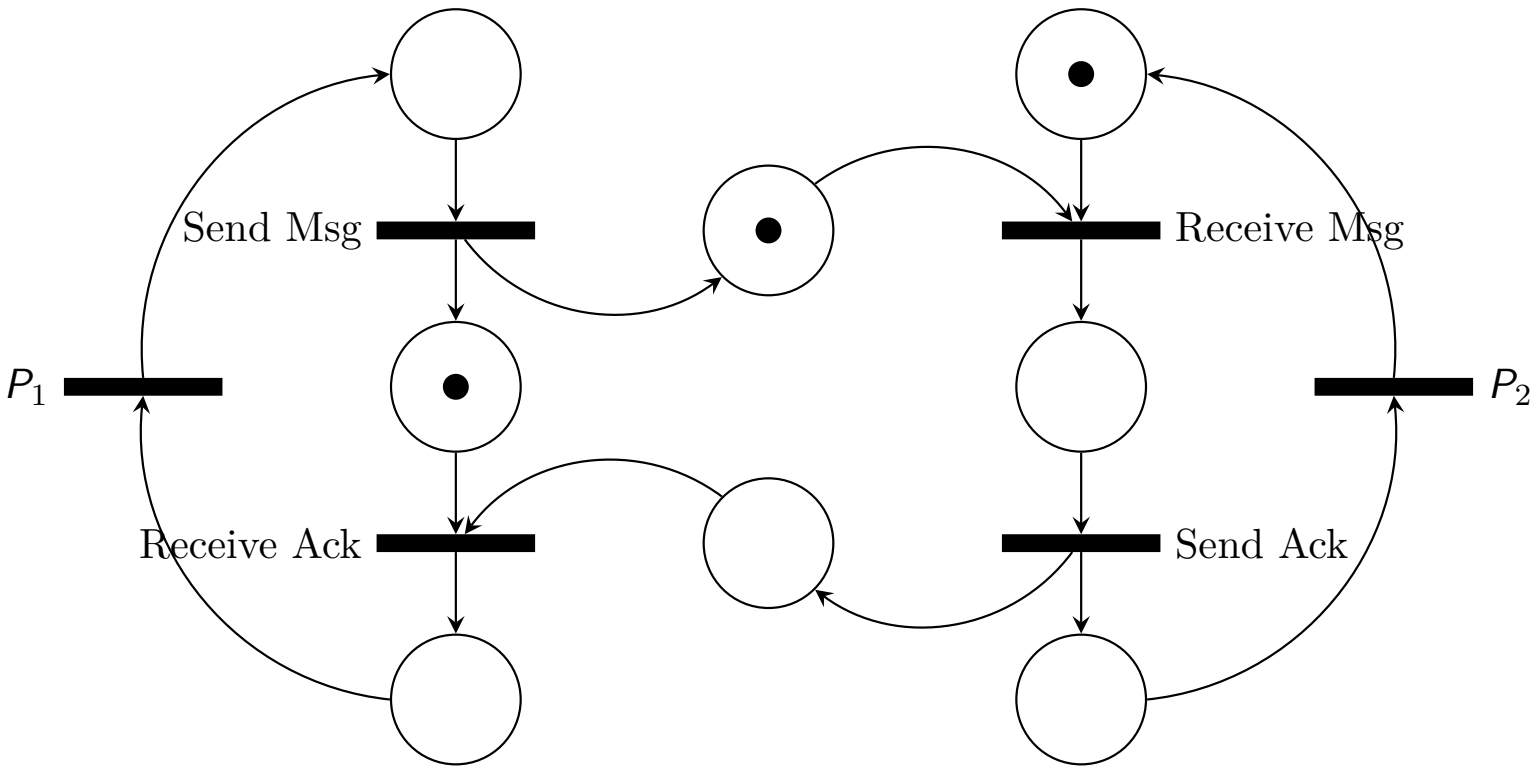


Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.

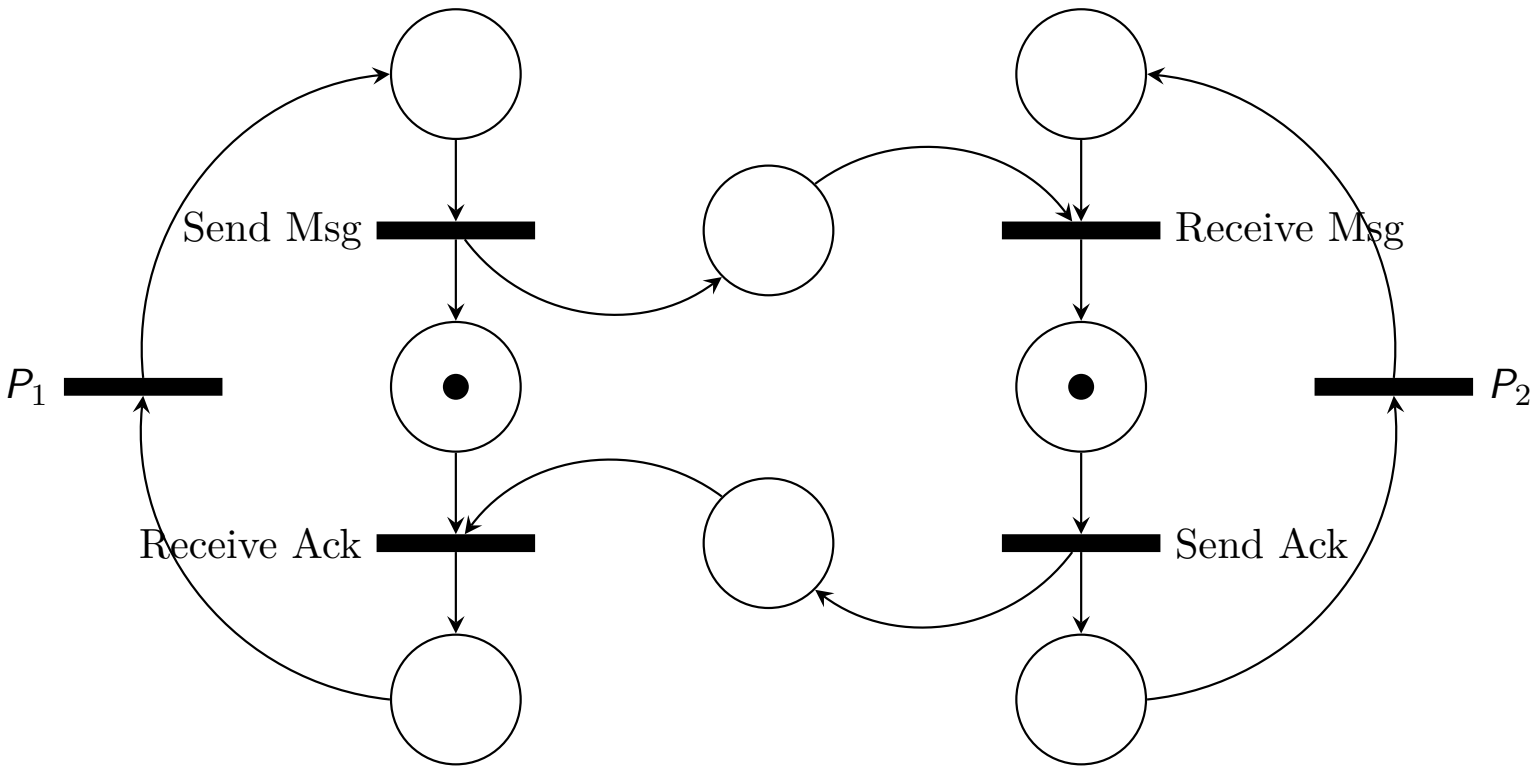


Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.

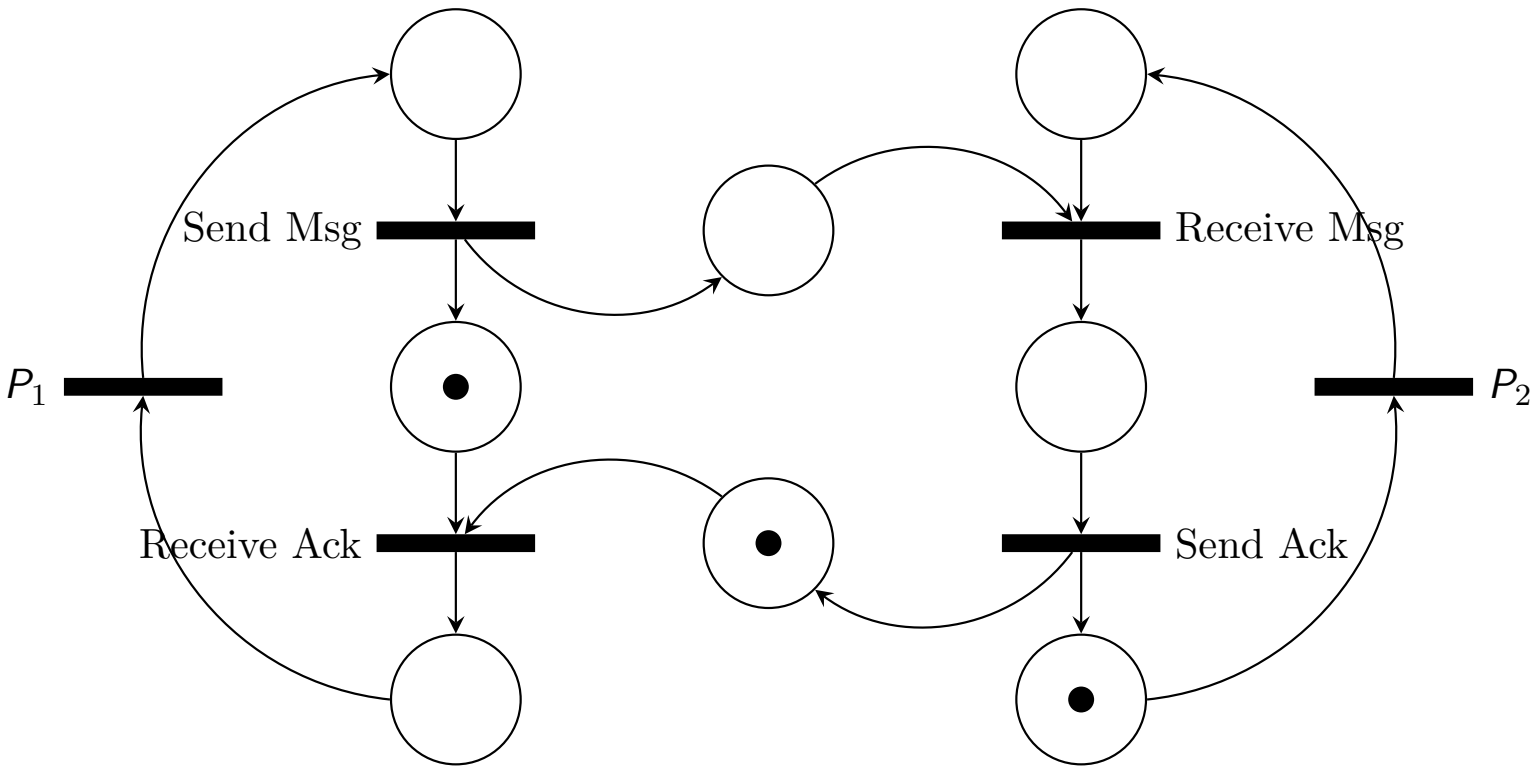


Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.

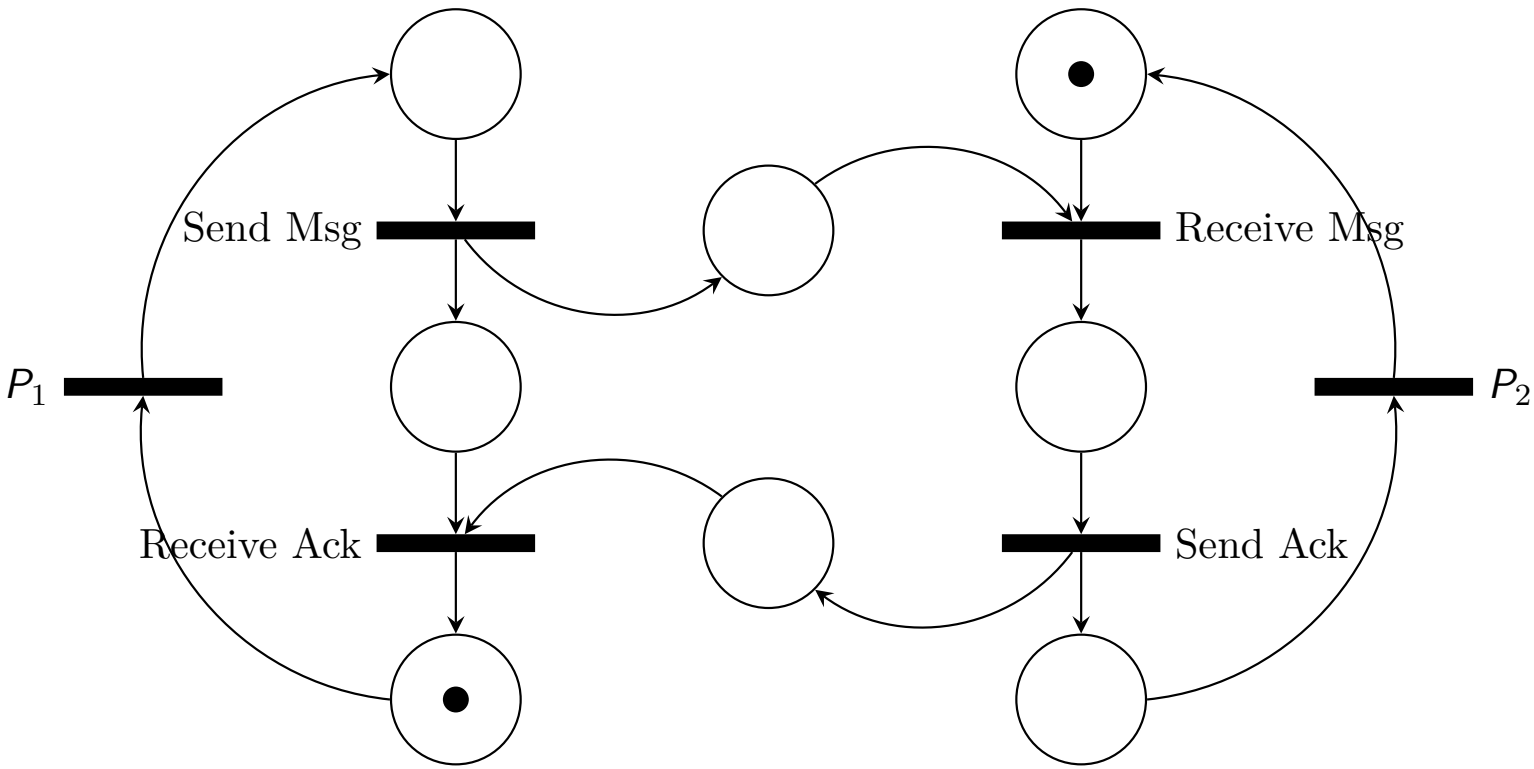


Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.

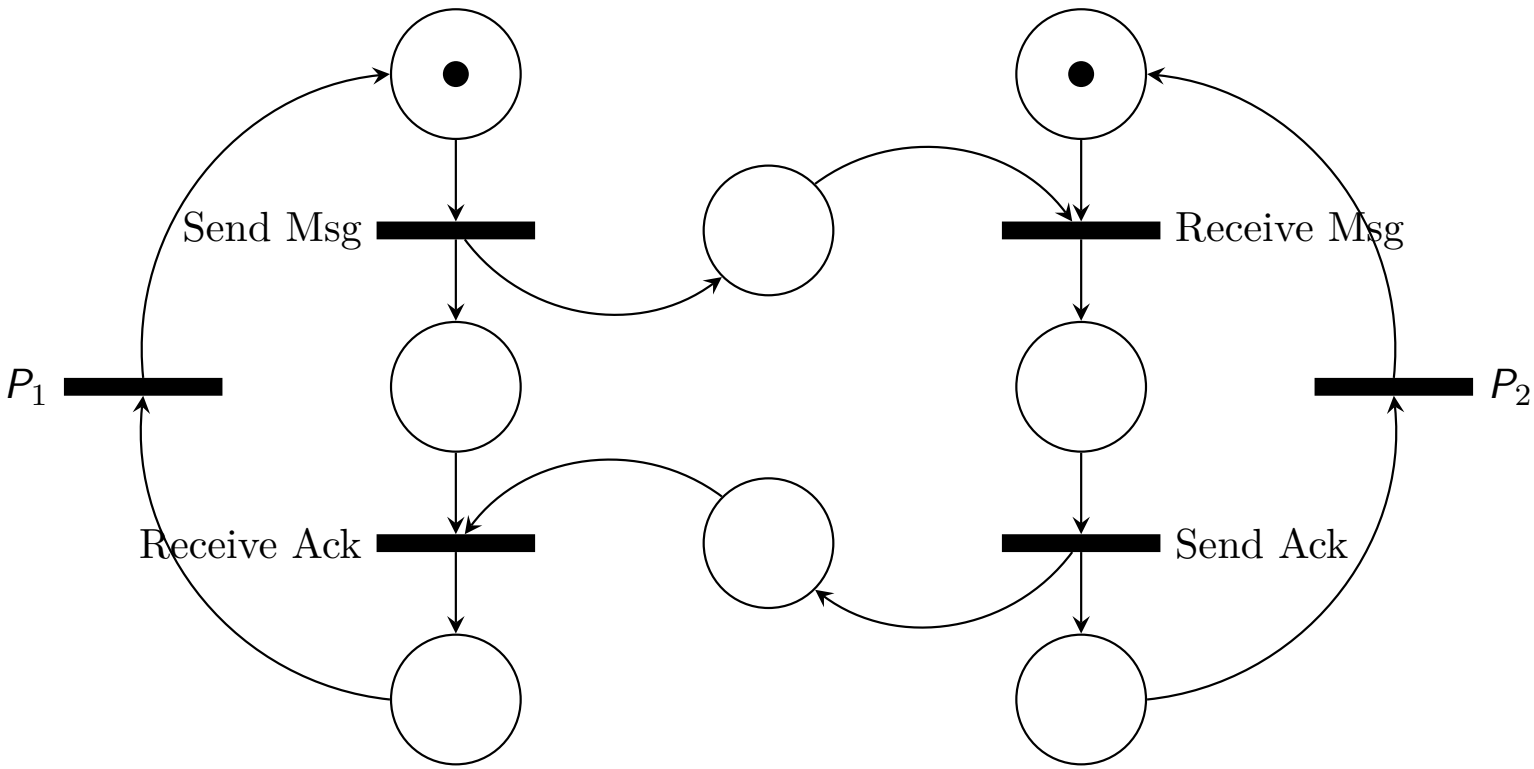


Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Przykład — protokół komunikacyjny

Rysunek 6 przedstawia sieć Petriego modelującą komunikację dwóch procesów z użyciem protokołu, w którym odbiorca potwierdza otrzymanie komunikatu od nadawcy.



Modelowanie protokołu komunikacyjnego z użyciem sieci Petriego (źródło: [6])

Sieci Petriego

Własności

Sieci Petriego mają szereg własności, których udowodnienie wykazuje, że oprogramowanie przez nie modelowane posiada określone atrybuty [7, 6]. Dzielą się one na dwie kategorie: *strukturalne* i *behawioralne*. Do tej ostatniej kategorii należą, między innymi:

- osiągalność (ang. *reachability*),
- żywotność (ang. *liveness*),
- ograniczoność (ang. *boundedness*),
- zachowawczość (ang. *conservation*).

Sieci Petriego

Osiągalność

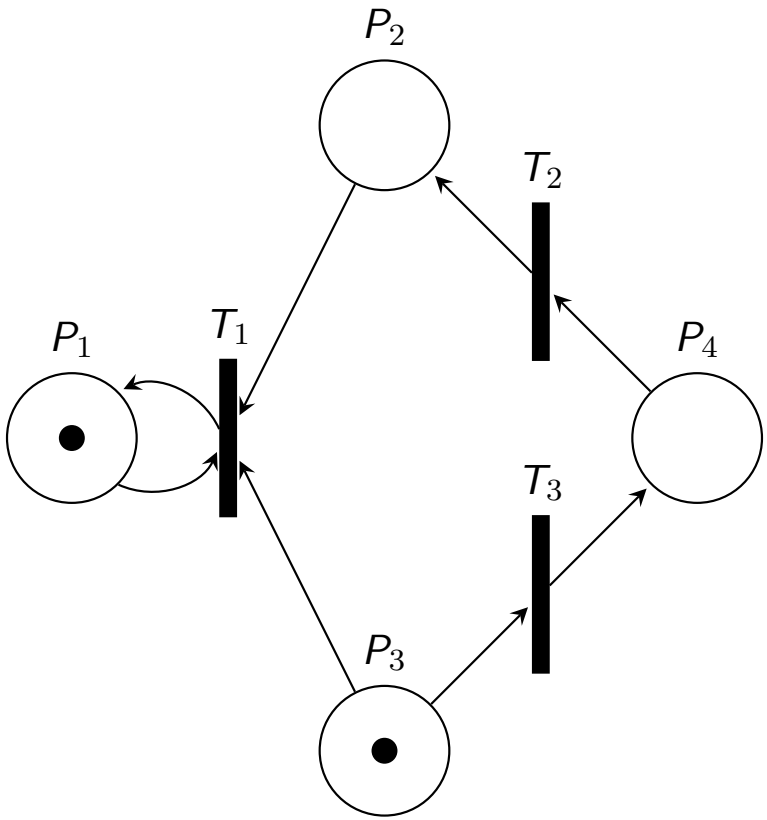
Osiągalność oznacza, że możliwe jest przejście z określonego stanu (znakowania) M_0 do wskazanego stanu M. Przykładowo w sieci przedstawionej na Rysunku 7 możliwe jest przejście ze stanu $M_0 = (1, 0, 1, 0)$ do stanu $M = (1, 1, 0, 0)$ poprzez odpalenie przejść T_3 i T_2:

$$M_0 = (1, 0, 1, 0) \xrightarrow{T_3} M_1 = (1, 0, 0, 1) \xrightarrow{T_2} M = (1, 1, 0, 0)$$

Kolejne pozycje w znakowaniu odpowiadają miejscom P_1, P_2, P_3 i P_4, a ich wartość opisuje, czy dane miejsce zawiera (1), czy nie (0) token.
Wykazanie osiągalności danego stanu pozwala udowodnić, że oprogramowanie w określonych warunkach może znaleźć się w pożądanym lub nie stanie, a także, że określony kod nie jest kodem martwym.

Sieci Petriego

Osiągalność



Ilustracja własności osiągalności w sieci Petriego (źródło: [6])

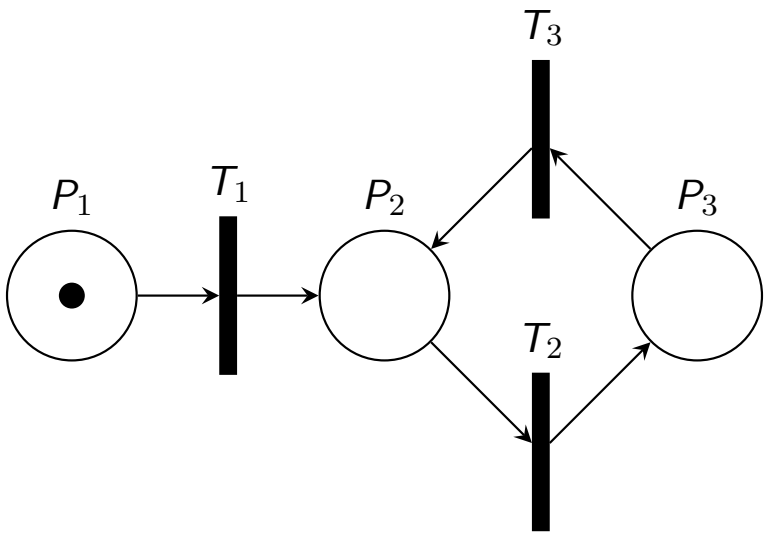
Sieci Petriego

Żywotność

Żywotność oznacza, że z każdego znakowania osiągalnego ze znakowania początkowego można odpalić (ostatecznie) każde przejście w sieci. W przełożeniu na oprogramowanie, ta własność gwarantuje, że nie będą występowały w nim zakleszczenia (ang. *deadlocks*), choć jej brak nie musi oznaczać, że się one pojawią. Sieć Petriego z Rysunku 8 nie jest żywa, bo przejście T_1 może być odpalone tylko ze znakowania, w którym token jest początkowo w miejscu P_1. Jednakże proszę zauważyć, że w fragmencie grafu składającym się z przejść T_2 i T_3 oraz miejsc P_2 i P_3 żywotność jest zapewniona. Mimo braku żywotności w całej sieci, w modelowanym przez nią oprogramowaniu nie wystąpi zakleszczenie.

Sieci Petriego

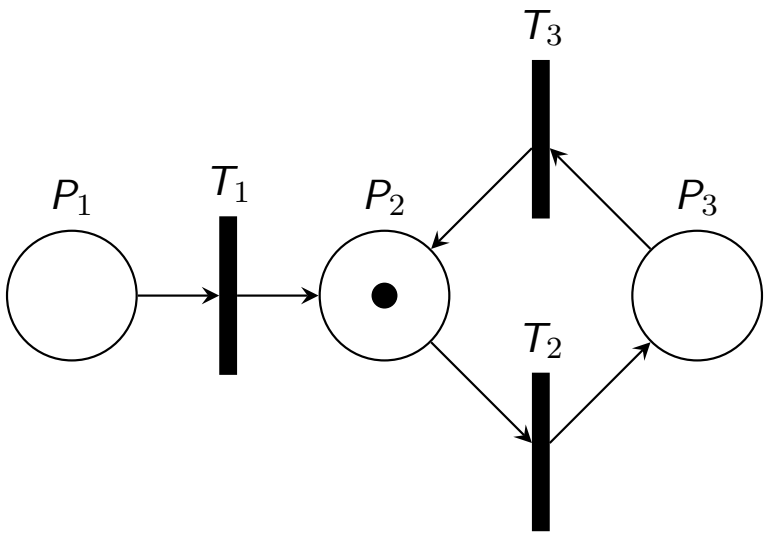
Żywotność



Ilustracja własności żywotności w sieci Petriego (źródło: [6])

Sieci Petriego

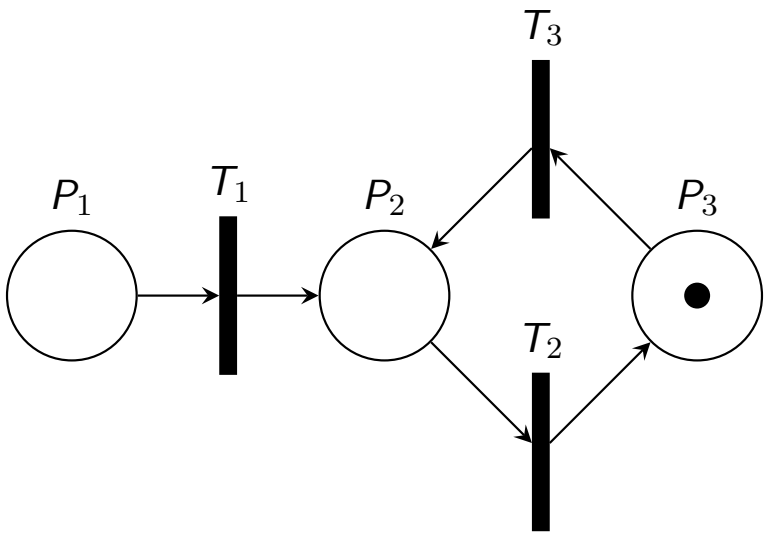
Żywotność



Ilustracja własności żywotności w sieci Petriego (źródło: [6])

Sieci Petriego

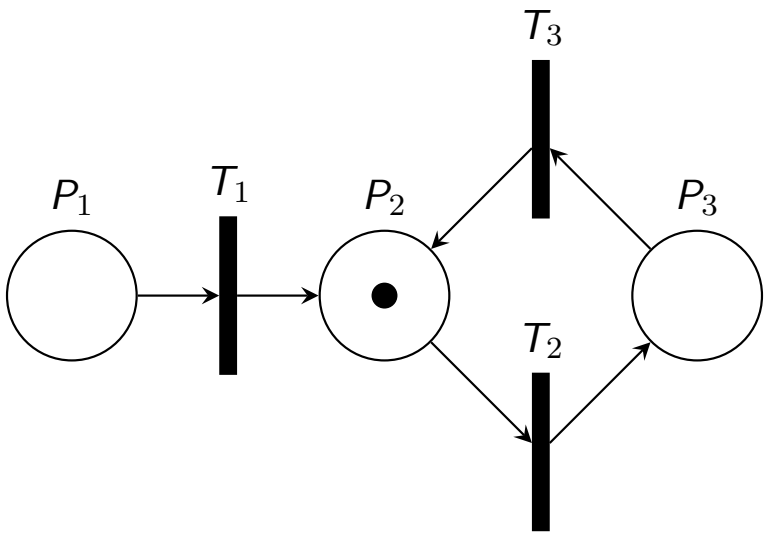
Żywotność



Ilustracja własności żywotności w sieci Petriego (źródło: [6])

Sieci Petriego

Żywotność



Ilustracja własności żywotności w sieci Petriego (źródło: [6])

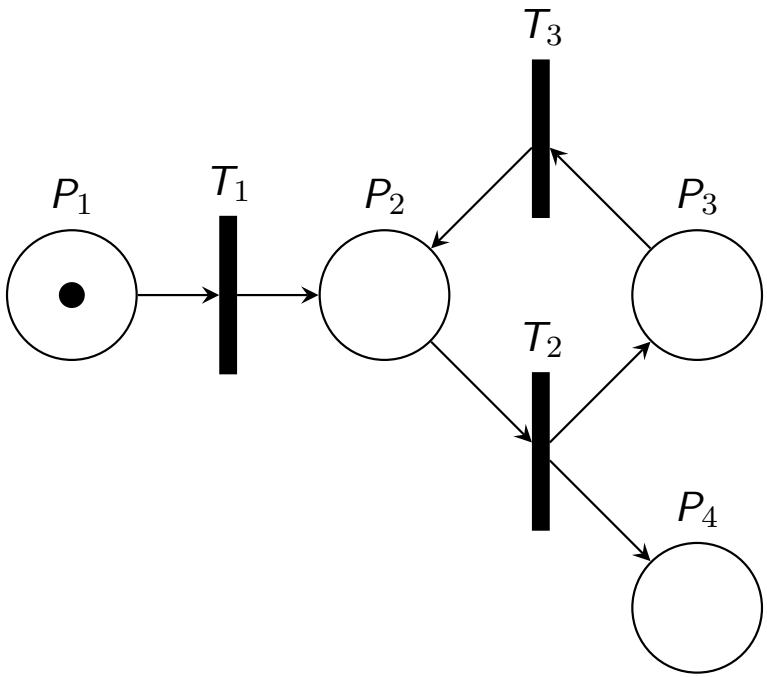
Sieci Petriego

Ograniczoność

Tokeny w sieci Petriego reprezentują zasoby systemu komputerowego. Ograniczoność oznacza, że liczba tokenów w dowolnym miejscu nie może rosnąć w nieskończoność. Jeśli sieć Peteriego *nie ma* własności ograniczoności, to oznacza, że w oprogramowaniu, które ona modeluje dochodzi do wycieku zasobów, np. w postaci obszarów pamięci, które są dynamicznie przydzielane, ale nie zwalniane, albo że dochodzi do przekroczenia pojemności (ang. *overflow*) buforów. Rysunek 9 przedstawia sieć Petriego modelującą oprogramowanie, gdzie takie problemy występują — w miejscu P_4 można umieścić nieskończoną liczbę tokenów.

Sieci Petriego

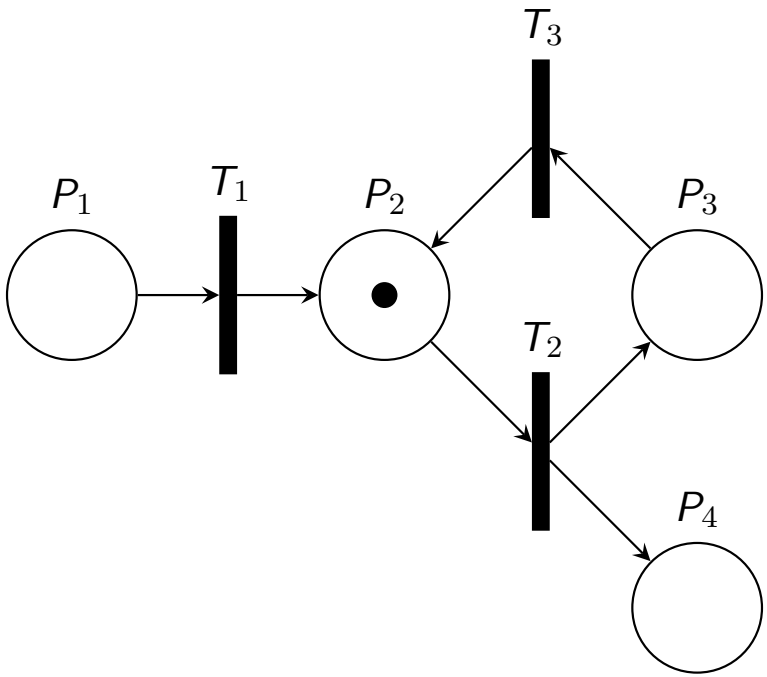
Ograniczoność



Ilustracja własności ograniczoności w sieci Petriego (źródło: [6])

Sieci Petriego

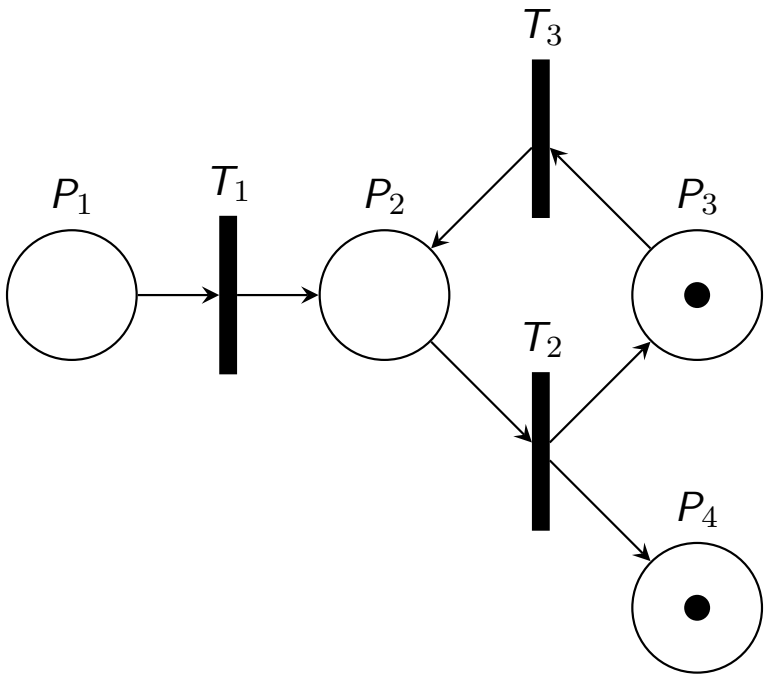
Ograniczoność



Ilustracja własności ograniczoności w sieci Petriego (źródło: [6])

Sieci Petriego

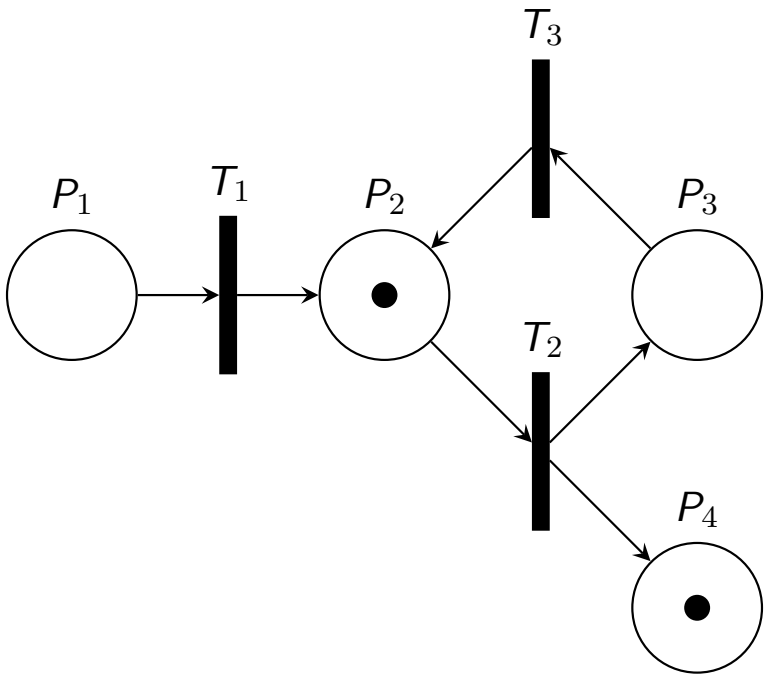
Ograniczoność



Ilustracja własności ograniczoności w sieci Petriego (źródło: [6])

Sieci Petriego

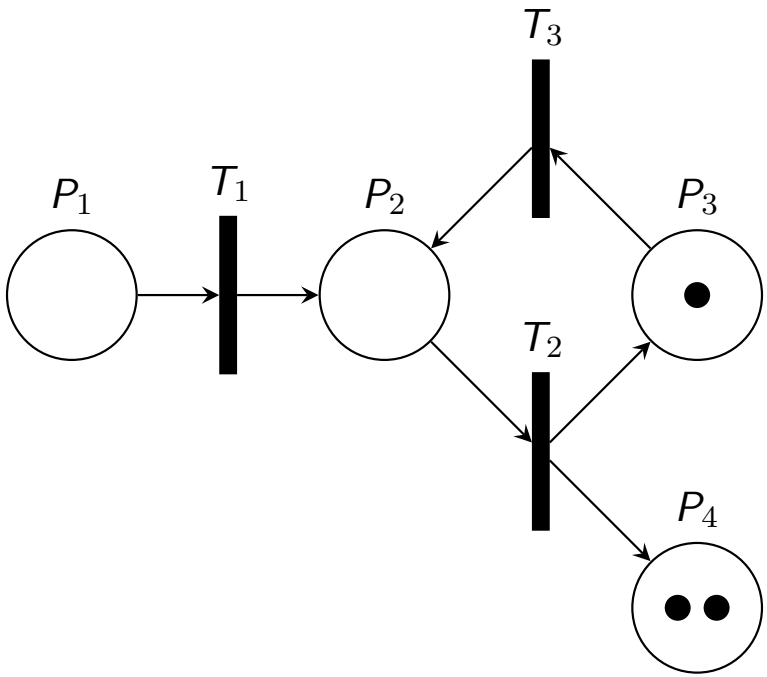
Ograniczoność



Ilustracja własności ograniczoności w sieci Petriego (źródło: [6])

Sieci Petriego

Ograniczoność



Ilustracja własności ograniczoności w sieci Petriego (źródło: [6])

Sieci Petriego

Zachowawczość

Zachowawczość oznacza, że w sieci Petriego występuje stała liczba tokenów. Omawiana wcześniej sieć z Rysunku 6 nie ma tej własności. Własności sieci Petiego dowodzi się z użyciem aparatu matematycznego z teorii grafów i powiązanego z nią rachunku macierzowego.

Zastosowania w przemyśle

Jedną z nowszych metod formalnych jest oparta na logice temporalnej ▸ TLA+, opracowana przez Lesliego Lamporta. Autor opracował dla niej szereg materiałów i narzędzi, włączając w to kurs ▸ wideo. Metoda ta została użyta między innymi do opracowania na zlecenie ESA wersji systemu operacyjnego czasu rzeczywistego, która okazała się ostatecznie dziesięciokrotnie mniejsza od swojej poprzedniczki, która była zainstalowana w sondzie biorącej udział w misji Rosetta [3]. Tej samej metody użyli pracownicy firmy Amazon do zaprojektowania i zaimplementowania serwisów webowych [8].
Specyfikacja Z z kolei została użyta w firmie Logica do weryfikacji elektronicznego systemu gotówkowego, bazującego na inteligentnych kartach (ang. *smart-cards*). Ta sama metoda stała się podstawą procesu tworzenia oprogramowania w firmie Rolls Royce and Associates (RRA).

Zastosowania w przemyśle

Najbardziej znanym przykładem zastosowania języka CPS jest opracowany na jego bazie język programowania *Occam*, który był stosowany w *transputerach* produkowanych przez firmę Inmos. Miał on także wpływ na projekt innych języków programowania, takich jak Go, Erlang, Clojure oraz bibliotek, takich jak ▸ Open MPI. Dodatkowo firma Altran UK (dawniej Praxis High Integrity Systems) zastosowała CSP do weryfikacji oprogramowania dla organu certyfikacji bezpiecznych kart inteligentnych [9].
Do firm korzystających, a także opracowujących metody formalne (np. VDM — *Vienna Development Method*) i oparte na nich procesy wytwarzania oprogramowania (np. metoda *Cleanroom*) należy IBM. Przykładem innych firm stosujących podejście formalne do wytwarzania oprogramowania są Airbus oraz ▸ Galois.

Zastosowania w przemyśle

Warto zaznaczyć, że wspierany przez takie języki programowania jak Haskell, Erlang, Elixir i Scala, paradygmat funkcyjny też bazuje na aparacie matematycznym, nazwanym *rachunkiem lambda*.
Dodatkowo, działanie analizatorów statycznych kodu, takich jak Frama-C lub ikos oparte jest na metodach formalnych.
Główną przeszkodą w stosowaniu metod formalnych, poza branżą oprogramowania o zastosowaniach krytycznych, jest brak personelu o odpowiednich kwalifikacjach i koszty ewentualnych szkoleń. Nawet w branży systemów krytycznych nie zawsze są one stosowane do tworzenia całości oprogramowania. Czasem ich użycie ogranicza się tylko do jego najistotniejszych fragmentów. Pomocą w ich wykorzystaniu są narzędzia programowe, które pozwalają np. zautomatyzować proces przeprowadzania dowodów matematycznych.

Źródła I


Gerard O'Regan. *Concise Guide to Formal Methods. Theory, Fundamentals and Industry Applications*. Springer, Cham, Switzerland, 2017.


Jon Bentley. *Perełki oprogramowania*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1992.


Eric Verhulst, Raymond E. Boute, José Miguel Sampaio Faria, Bernhard H.C. Sputh i Mezhuyev Vitaliy. *Formal Development of Network-Centric RTOS. Software Engineering for Reliable Embedded Systems*. Springer, New York, 2011.


Jerzy Nawrocki. Metody formalne. URL: http://smurf.mimuw.edu.pl/external_slides/Metody_formalne/Metody_formalne.html (termin wizyty 30. 11. 2024).

Źródła II

- Charles Antony Richard Hoare. Communicating Sequential Processes. URL: http://www.usingcsp.com/cspbook.pdf (termin wizyty 03. 12. 2024).
- Petri Nets. URL: https://ptolemy.berkeley.edu/projects/embedded/research/hsc/class/ee249/lectures/l7-PetriNets.pdf (termin wizyty 02. 12. 2024).
- Jędrzej Ułasiewicz. Sieeci Petriego. URL: http://jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl/ProgramowanieWspolbiezne/wyklad/Sieci-Petriego15.pdf (termin wizyty 02. 12. 2024).
- Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker i Michael Daerduff. How Amazon Web Services Uses Formal Methods. 2015. DOI: 10.1145/2699417. URL: https://cacm.acm.org/research/how-amazon-web-services-uses-formal-methods/ (termin wizyty 01. 12. 2024).

Źródła III

Anthony Hall i Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. 2002. URL: https://www.anthonyhall.org/c_by_c_secure_system.pdf.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!