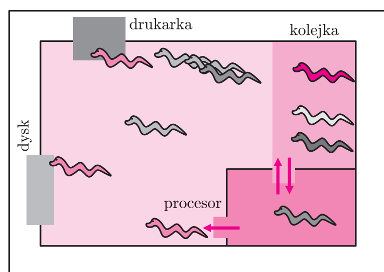
# O topologii programów współbieżnych

*Maria DONTEN**

Czy informatycy zajmują się dowodzeniem twierdzeń? Zdarza się, że poprawność programu, czy pomysłu na program, jest nieoczywista. Wtedy potrzebne są metody, które pozwolą przeprowadzić ścisły dowód, że zaprojektowany algorytm zadziała tak, jak powinien. W niektórych zagadnieniach pomysły na dowód sięgają w głąb matematyki.

Przyjrzyjmy się paradygmatowi programowania współbieżnego. Jeśli chcemy rozwiązać duży problem obliczeniowy, to dobrym pomysłem jest podzielenie go na podproblemy. Dzięki temu, jeśli w obliczeniach pojawi się błąd, wystarczy powtórzyć jedynie część, która nie została wykonana poprawnie. Jeśli mamy odpowiedni sprzęt, np. kilka komputerów lub komputer z kilkoma procesorami, możemy przyspieszyć obliczenia, pracując nad wieloma podproblemami w tym samym czasie. W ten sposób podejście sekwencyjne zastępujemy przez współbieżne. Opisana idea współbieżności jest bardzo ogólna – można ją stosować nie tylko do obliczeń komputerowych, lecz także do rozwiązywania dowolnych problemów. Szczegółami zajmiemy się jednak tylko w przypadku programowania.




Rys. 1. Życie codzienne procesów.

Podstawowym pojęciem tej dziedziny jest proces sekwencyjny (zwykle nazywany po prostu procesem), czyli realizacja programu w pewnym środowisku lub wstrzymana realizacja w oczekiwaniu na pewne zdarzenie. Realizacja programu to ciąg wykonań kolejnych instrukcji. Często wykonanie jakiejś operacji wymaga sprzyjających warunków. Jeśli trzeba na nie zaczekać, wykonanie programu jest na chwilę wstrzymywane. Proces to byt bardzo abstrakcyjny – czy „ciąg wykonań instrukcji" można sobie jakoś wyobrazić, narysować? Otóż można: wystarczy do każdego ciągu instrukcji dodać stworzonko, które odpowiada za jego wykonanie, czyli wędruje po wnętrzu komputera, realizując kolejne zadania w odpowiednich miejscach. Na przykład stoi w kolejce do drukarki, zapisuje coś na dysku, czeka na przydział czasu do obliczeń na procesorze lub właśnie wykonuje obliczenia (rys. 1). Ten obrazek pozwala mniej więcej wyobrazić sobie, co się odbywa w systemie operacyjnym podczas wykonywania programów. Jego wersje z nieco większą liczbą szczegółów przydają się wszystkim studentom, którzy zaliczają laboratorium z programowania współbieżnego...

Dążymy do tego, żeby rozwiązywać problemy programistyczne, rozdzielając pracę między kilka procesów działających w tym samym czasie. Dokładniej, dwa procesy nazwiemy współbieżnymi, jeśli jeden z nich rozpoczął działanie po rozpoczęciu, ale przed zakończeniem działania drugiego. Program współbieżny składa się z kilku procesów współbieżnych, wykonujących elementy pewnego zadania. I tu pojawiają się trudności... Jeśli procesy pracują razem nad jednym problemem, korzystają ze wspólnych danych, potrzebują nawzajem swoich częściowych wyników, to muszą się komunikować, żeby nie doszło do żadnych nadużyć.

Jak procesy się komunikują? Mają swoje sposoby, zależne od języka programowania lub abstrakcyjnej notacji używanej do opisu programów współbieżnych. Na przykład, nadają komunikat do konkretnego adresata, który musi w odpowiedniej chwili nastawić się na odbiór (CSP), albo umawiają się na randki (Ada), albo wrzucają dane do worka i wyciągają je według szablonów (Linda)...

Najprostszym przykładem możliwości nadużycia jest problem sekcji krytycznej, czyli ustalenia, który proces uzyska dostęp do zasobu, z którego w danej chwili może korzystać tylko jeden proces. Przyjrzyjmy się prostemu programowi, w którym działają dwa procesy, $P$ i $Q$. Każdy z nich wykonuje następujący ciąg czynności: wczytuje wartość zmiennej globalnej $v$ na swoją zmienną lokalną, zwiększa jej wartość o 1 i zapisuje wynik na zmiennej $v$.

```
                var v: integer; v := 0;
process P;                    process Q;
var P.i, P.x : integer;       var Q.i, Q.x : integer;
P.x := v;                     Q.x := v;
P.x := P.x + 1;               Q.x := Q.x + 1;
v := P.x;                     v := Q.x;
end;                          end;
```

Oczywiście, procesy danego programu nie muszą wykonywać tego samego (lub symetrycznego) kodu, tak jak w przykładzie.

Zmienna $v$ ma początkowo wartość 0, każdy proces zwiększa jej wartość o 1, więc po zakończeniu działania programu $v$ powinna mieć wartość 2. Czy na pewno ten program tak właśnie zadziała? Często tak, na przykład w ekstremalnym przypadku, kiedy proces $P$ rozpocznie działanie tuż przed wykonaniem końcowej operacji procesu $Q$, ale nie zawsze. Sytuacja komplikuje się, jeśli procesy wykonują te same instrukcje w prawie tym samym czasie. Powiedzmy, że $P$ wczytuje wartość zmiennej $v$ i, zanim zdąży wykonać kolejną czynność, $Q$ rozpoczyna działanie i również wczytuje wartość $v$ – zmienne lokalne $P.x$ i $Q.x$ mają wartość 0. Jeśli

*doktorantka, Instytut Matematyki UW

teraz $P$ zwiększy wartość $P.x$ i zapisze ją na $v$, a potem $Q$ wykona te same instrukcje, to zmienna $v$ będzie miała wartość 1, chociaż każdy z procesów wykonał dodawanie! Wynik jest niezgodny z oczekiwaniami. Powstał błąd, ponieważ procesy nie uzgodniły, który najpierw zmodyfikuje zmienną globalną. Proces modyfikacji składa się z kilku instrukcji: wczytania, zwiększenia i zapisania. Język programowania nie zapewnia, że te wszystkie czynności zostaną wykonane w jednym, nieprzerwanym bloku – zadbanie o to jest zadaniem programisty!

Okazuje się, że ominięcie tego problemu wcale nie jest łatwe... Właściwie jest niemożliwe bez dodatkowych mechanizmów służących do synchronizacji procesów, zdefiniowanych na poziomie systemu operacyjnego.

Nawet z tego prostego przykładu widać, że sprawdzenie, czy program współbieżny jest poprawny, może być bardzo trudne. Trzeba sprawdzić, czy działanie programu jest zgodne z oczekiwaniami dla każdego przeplotu, czyli dla każdej sekwencji wykonań instrukcji procesów biorących udział w programie – począwszy od przypadków, w których czasy wykonania procesów prawie się nie pokrywają, aż do takich, w których procesy wykonują na zmianę bardzo małe fragmenty kodu. Takich przeplotów może być bardzo dużo, w rzeczywistych problemach często nieskończenie wiele... A oprócz bezpieczeństwa programu, czyli zapewnienia, że w każdym momencie w sekcji krytycznej będzie co najwyżej jeden proces, należy zadbać jeszcze o inne własności. Między innymi nie można zagłodzić procesu, czyli spowodować, że nie będzie mógł nigdy wykonać pewnej instrukcji swojego kodu, bo, na przykład, inne procesy będą się zawsze przed nim wpychać do kolejki. Nie można również dopuścić do zakleszczenia – sytuacji, w której grupa procesów oczekuje na zdarzenie, które może spowodować tylko proces z tej grupy.

Pytanie, czy dla danego problemu istnieje poprawny program współbieżny, który go rozwiązuje, wygląda na jeszcze trudniejsze niż sprawdzenie poprawności napisanego kodu. Jednak dość niedawno powstało matematyczne kryterium, które pozwala spojrzeć na to pytanie z zupełnie innej strony, a często również udzielić kompletnej odpowiedzi. Oczywiście nie dla zupełnie dowolnych problemów. Interesująca nas klasa to problemy (lub zadania) decyzyjne (ang. *decision tasks*), w których współbieżność jest w pewien sposób wbudowana w sformułowanie zadania. Inaczej mówiąc, takie, których istotą jest dokonanie pewnych uzgodnień pomiędzy procesami. Procesy uczestniczące w zadaniu dostają dane wejściowe, zazwyczaj liczby, wykonują pewien algorytm i podają wynik, który musi być zgodny ze specyfikacją problemu. Zobaczmy kilka przykładów.

W literaturze informatycznej w języku polskim często pozostawiane są angielskie nazwy problemów i algorytmów, co ułatwia porównanie z literaturą anglojęzyczną. Również tutaj stosujemy tę konwencję.

Najprostszy w sformułowaniu (ale nie w rozwiązaniu) problem to *Consensus*. Uruchamiamy pewną liczbę procesów, każdy z nich dostaje wartość wejściową 0 lub 1. Jako wynik wszystkie procesy mają podać tę samą wartość $x$, będącą jedną z wartości, które dostały na wejściu. W szczególności, jeśli wartości wejściowe były równe, to jest tylko jeden dopuszczalny wynik; w pozostałych przypadkach procesy mają większą dowolność w wyborze wyniku. Może się zdarzyć, że nie wszystkie procesy, które rozpoczną działanie, zakończą się poprawnie i podadzą wynik. Wtedy chcemy wymagać, żeby procesy, w których nie wystąpił błąd, zachowywały się zgodnie z założeniami zadania – wynik ma być zgodny ze specyfikacją dla przypadku, w którym uruchomione zostałyby tylko te procesy, które zadziałały bezbłędnie.

Dalej przyda się nam problem *Binary Consensus*, będący ograniczeniem powyższego do przypadku dwóch procesów, i jego modyfikacja, *Quasi-Consensus*. Zmiana polega na dopuszczeniu jeszcze jednego wyniku: jeśli dane wejściowe procesów $P$ i $Q$ są różne, to oprócz odpowiedzi $P \mapsto 0$, $Q \mapsto 0$ i $P \mapsto 1$, $Q \mapsto 1$ za prawidłową uznajemy również $P \mapsto 1$, $Q \mapsto 0$ (ale odwrotnej już nie!).

Bardziej złożonym zadaniem jest *Unique-Id* – w systemie działa $n$ procesów, wartości wejściowe są nieistotne (np. wszystkie są zerami), procesy muszą wybrać różne wartości ze zbioru $\{0, \ldots, n-1\}$. Ciekawy przykład „z życia" to *Renaming* – procesy dostają różne wartości z dużego zbioru $\{0, \ldots, M\}$, zwracają różne wartości z mniejszego zbioru $\{0, \ldots, K\}$. To zadanie można przetłumaczyć na problem układania bezkolizyjnych tras dla samolotów. Wystarczy podawać numery identyfikacyjne lotów na wejściu, a wyniki potraktować jako numery korytarzy powietrznych, których zazwyczaj jest istotnie mniej niż wszystkich samolotów.

Jak wygląda precyzyjny matematyczny opis interesujących nas zadań? Zestawy wartości wejściowych i wyjściowych będziemy zapisywać jako wektory – mogą w nich występować liczby lub wartość nieokreślona, oznaczana $\perp$ (nie dopuszczamy wektora złożonego tylko z wartości nieokreślonych; to kwestia notacji). W wektorze

wejściowym wartość $\perp$ na $i$-tym miejscu oznacza, że $i$-ty proces nie bierze udziału w przebiegu programu, natomiast w wektorze wyjściowym informuje, że $i$-ty proces nie zakończył działania poprawnie – w ogóle nie brał udziału w danym przebiegu lub zakończył się z błędem. Zbiory wektorów wejściowych i wyjściowych oznaczamy odpowiednio $\mathcal{I}$ i $\mathcal{O}$. Potrzebna jest nam jeszcze relacja $\Delta \subseteq \mathcal{I} \times \mathcal{O}$ między tymi zbiorami, określająca, które wyniki uznajemy za poprawne dla różnych wektorów wejściowych. Zwykle będziemy patrzeć na $\Delta$ jako na funkcję z $\mathcal{I}$ w zbiór wszystkich podzbiorów $\mathcal{O}$. To już prawie wystarczy – trzeba jeszcze założyć, że $\mathcal{I}$, $\mathcal{O}$ i $\Delta$ mają pewne dobre własności. Zbiory wektorów mają być zamknięte na prefiksy: dla każdego wektora $\alpha \in \mathcal{I}$ w zbiorze $\mathcal{I}$ muszą być również wszystkie wektory, które można otrzymać z $\alpha$, zastępując niektóre liczby przez $\perp$; to samo dla $\mathcal{O}$. Natomiast od $\Delta$ wymagamy, żeby w obrazie wektora $\alpha$ były tylko wektory, które mają $\perp$ na tych samych pozycjach co $\alpha$.

Warto spróbować zinterpretować te założenia, pamiętając, jakie znaczenie ma $\perp$ w wektorach zbiorów $\mathcal{I}$ i $\mathcal{O}$.

Trójka $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$, której elementy spełniają powyższe założenia, to kompletny opis zadania. Problem *Binary Consensus* możemy więc sformułować, definiując

$$\mathcal{I} = \{(0,\perp),(1,\perp),(\perp,0),(\perp,1),(0,0),(0,1),(1,0),(1,1)\},$$
$$\mathcal{O} = \{(0,\perp),(1,\perp),(\perp,0),(\perp,1),(0,0),(1,1)\},$$

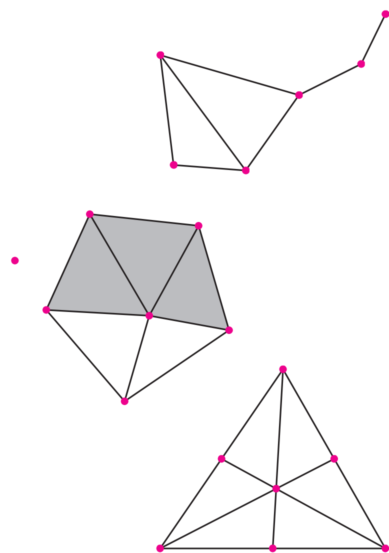
oraz $\Delta(\alpha) = \{\alpha\}$, jeśli $\alpha$ zawiera $\perp$, a w pozostałych przypadkach

$$\Delta((0,0)) = \{(0,0)\}, \quad \Delta((1,1)) = \{(1,1)\},$$
$$\Delta((1,0)) = \Delta((0,1)) = \{(0,0),(1,1)\}.$$

Opis zadania *Quasi-Consensus* otrzymujemy z powyższego, zmieniając tylko wartość $\Delta$ dla wektorów $(0,1)$ i $(1,0)$:

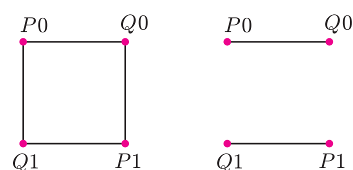
$$\Delta((1,0)) = \Delta((0,1)) = \{(0,0),(1,1),(1,0)\}.$$

Uzyskaliśmy prosty, kombinatoryczny sposób opisu problemów, które chcemy rozwiązywać. Jednak dla trudnych zadań, w których uczestniczy wiele procesów, a takie zwykle pojawiają się w życiu, struktury kombinatoryczne są duże, bardzo złożone, trudno coś na ich temat udowodnić... A gdyby spojrzeć na te obiekty „z daleka”, spróbować dostrzec ich kształt, jakieś charakterystyczne cechy, które pozwolą je lepiej zrozumieć bez analizowania wszystkich szczegółów? I tu wreszcie wkracza topologia!

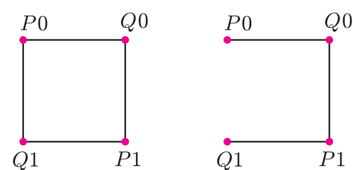


Rys. 2. Kilka kompleksów symplicjalnych.

Okazuje się, że zbiory $\mathcal{I}$ i $\mathcal{O}$ opisują kompleksy symplicjalne – przestrzenie złożone ze skończenie wielu sympleksów (punktów, odcinków, trójkątów, czworościanów, ...) posklejanych porządnie, czyli wspólnymi ścianami. Kilka przykładów można obejrzeć na rysunkach na marginesie. Taki kompleks jest obiektem topologicznym, ale równocześnie można go zdefiniować kombinatorycznie – za pomocą skończonego zbioru informacji. Wystarczy umieścić w odpowiednio dużej przestrzeni $\mathbb{R}^n$ wszystkie wierzchołki wszystkich sympleksów, które mają się znaleźć w kompleksie, i powiedzieć, które pary łączymy odcinkiem, potem które z powstałych w ten sposób trójkątów mają być wypełnione, itd. Właśnie jako takie zestawy danych możemy traktować zbiory $\mathcal{I}$ i $\mathcal{O}$. Każdemu z nich przypisujemy kompleks, którego wierzchołkami są wektory zawierające tylko jedną wartość różną od $\perp$ (na rysunkach będą one oznaczane przez nazwę procesu, któremu odpowiada wartość określona, i tę wartość). Odcinek łączący dwa wierzchołki należy do kompleksu, jeśli w zbiorze znajduje się wektor, który ma dokładnie dwie wartości różne od $\perp$ – takie, jak w wektorach będących wierzchołkami, i na tych samych miejscach. Analogicznie dodajemy do kompleksu sympleksy wyższych wymiarów – odpowiadają one wektorom z mniejszą liczbą $\perp$. Rysunki 3 i 4 przedstawiają kompleksy wejściowe i wyjściowe dla zadań *Binary Consensus* i *Quasi-Consensus*. Przekształcenie $\Delta$ przypisuje sympleksom z kompleksu $\mathcal{I}$ zbiory sympleksów z $\mathcal{O}$, zachowując wymiar. Wobec tego, niestety, zazwyczaj nie może być ono przekształceniem symplicjalnym, czyli takim, że obrazem każdego sympleksu jest sympleks, ale okaże się, że czasem jest w pewnym sensie bliskie takiemu.




Rys. 3. Kompleksy $\mathcal{I}$ i $\mathcal{O}$ dla zadania *Binary Consensus*.

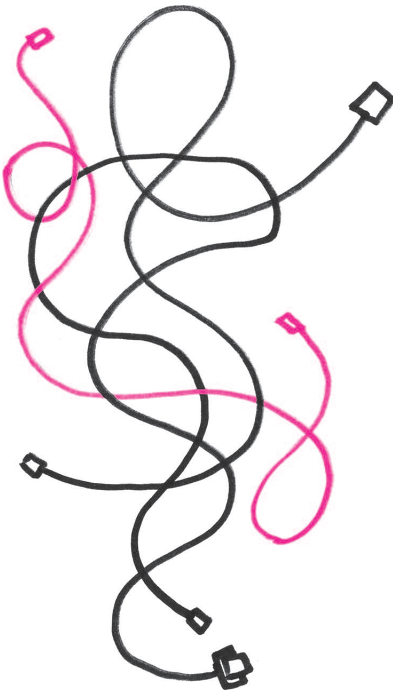



Rys. 4. Kompleksy $\mathcal{I}$ i $\mathcal{O}$ dla zadania *Quasi-Consensus*.

Jak wyrazić założenia, które przyjęliśmy dla $\mathcal{I}$, $\mathcal{O}$ i $\Delta$, w języku odpowiadających im kompleksów i przekształcenia?

Przyda się nam jeszcze jedna operacja na kompleksie symplicjalnym – podział kompleksu. Każdy sympleks wyjściowego kompleksu możemy rozbić na kilka mniejszych (dodając nowe wierzchołki), nawet na wiele różnych sposobów, ale żeby wynikiem był kompleks symplicjalny, rozbicie musi spełniać warunek dobrego sklejania sympleksów. Łatwo narysować, na przykład, podział barycentryczny, który powstaje przez dodanie i odpowiednie połączenie środków ciężkości wszystkich sympleksów. Najniższy kompleks na rysunku 2 to właśnie podział barycentryczny

trójkąta. Nośnikiem sympleksu w podziale nazywamy najmniejszy sympleks wyjściowego kompleksu, który go zawiera.

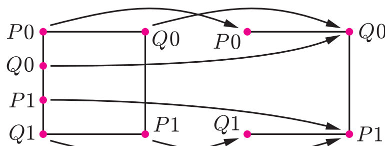


Czasami wierzchołki w kompleksie są pokolorowane; jak zwykle wymagamy, żeby wierzchołki połączone odcinkiem miały różne kolory. Wtedy, konstruując podział, chcemy, żeby kolorowanie można było rozszerzyć na dodane wierzchołki, i wybieramy pewne rozszerzenie. Na kompleksach $\mathcal{I}$ i $\mathcal{O}$ uwzględniamy kolorowanie pochodzące od oznaczeń wierzchołków – kolory to pary złożone z nazwy procesu i wartości wejściowej lub wyjściowej.

Te pojęcia pozwolą nam przyjrzeć się twierdzeniu, które opisuje zależność między istnieniem dobrego rozwiązania dla zadania a strukturą odpowiadających mu kompleksów symplicjalnych. Ale jakie rozwiązanie uznamy za dobre? Wymagamy oczywiście, żeby wyniki były zgodne z funkcją $\Delta$. Dodatkowo protokół, czyli algorytm wykonywany przez procesy, nie może pozwalać na nieskończone oczekiwanie – każdy proces po skończonej liczbie kroków musi zakończyć działanie, poprawnie lub z błędem (ang. *wait-free protocol*). Żeby można było tę definicję wykorzystać w ścisłym dowodzie, trzeba ją zapisać w języku automatów, ale my poprzestaniemy na intuicji.

Poniższe twierdzenie pochodzi z pracy Maurice'a Herlihy'ego i Nira Shavita, *The Topological Structure of Asynchronous Computability* (*Struktura topologiczna obliczeń rozproszonych*), z 1999 roku. Jak widać, pomysł nie jest stary, i można się spodziewać, że nie został jeszcze całkiem wyeksploatowany...

**Twierdzenie 1.** *Dla zadania $\langle\mathcal{I},\mathcal{O},\Delta\rangle$ istnieje protokół bez oczekiwania wtedy i tylko wtedy, gdy istnieje podział $\sigma(\mathcal{I})$ kompleksu $\mathcal{I}$ (z opisanym wyżej kolorowaniem) oraz przekształcenie symplicjalne $\mu:\sigma(\mathcal{I})\to\mathcal{O}$ zachowujące kolorowanie i takie, że dla każdego wierzchołka $s\in\sigma(\mathcal{I})$ punkt $\mu(s)$ leży w obrazie nośnika $(s,\mathcal{I})$ przy funkcji $\Delta$.*




Rys. 5. Działanie przekształcenia $\mu$ dla zadania *Quasi-Consensus*.

To oznacza, że jeśli chcemy udowodnić istnienie dobrego protokołu dla pewnego zadania, musimy znaleźć podział kompleksu wejściowego uwzględniający naturalne kolorowanie oraz odwzorowanie $\mu$ z tego podziału w $\mathcal{O}$, zgodne z warunkami zadania, czyli z funkcją $\Delta$. Przyjrzyjmy się, jak to wygląda dla zadania *Quasi-Consensus* (rys. 5).

Kompleksy $\mathcal{I}$ i $\mathcal{O}$ już znamy, a $\sigma(\mathcal{I})$ tworzymy, dodając na krawędzi $(P0, Q1)$ dwa punkty, o kolorach $P1$ i $Q0$. Przekształcenie $\mu$ przeprowadza krawędzie $(P0, Q0)$, $(P1, Q0)$, $(P1, Q1)$ na ich odpowiedniki w kompleksie wyjściowym. Trzy odcinki składające się na krawędź $(P0, Q1)$ również przechodzą przy $\mu$ na te krawędzie, zgodnie z kolorami wierzchołków. Łatwo sprawdzić, że warunek zgodności z $\Delta$ jest spełniony. Stąd dla zadania *Quasi-Consensus* istnieje protokół bez oczekiwania.

A co z zadaniem *Binary Consensus*? Spójrzmy jeszcze raz na rysunek 3 – kompleks wyjściowy jest niespójny. Warunek zgodności z $\Delta$ wymaga, żeby obie składowe $\mathcal{O}$ były w obrazie $\mu$. Ale $\mu$ musi być ciągłe, a $\mathcal{I}$ jest spójny, co prowadzi do sprzeczności. Dla tego zadania dobry protokół nie istnieje. A żeby to wykazać, wystarczyło przyjrzeć się bardzo pobieżnie strukturze topologicznej kompleksów!

W tym miejscu warto sobie przypomnieć o problemie sekcji krytycznej i sygnalizowanych trudnościach w jego ominięciu...

Można stawiać zarzuty, że twierdzenie daje wnioski czysto egzystencjalne – a co nam przyjdzie z wiedzy, że pewien program można napisać, jeśli nie wiemy, jak to zrobić? W rzeczywistości jest lepiej, niż mogłoby się na pierwszy rzut oka wydawać. Co prawda z samego twierdzenia nie możemy wiele wnioskować o postaci protokołu, ale dowód jest konstruktywny. Jego analiza pozwala na zrozumienie i zapisanie odpowiedniego kodu, chociaż zwykle nie w takiej postaci, jaka wydawałaby się najbardziej naturalna.

Użyliśmy twierdzenia 1 tylko do rozstrzygnięcia dwóch bardzo prostych problemów. Ale można je stosować do istotnie bardziej złożonych sytuacji i otrzymywać ważne rezultaty. To twierdzenie pozwoliło, między innymi, na zbadanie zadania *Renaming*. W pewnych przypadkach, dla zbioru wyników niewiele większego od zbioru procesów, istnienie protokołu bez oczekiwania było wcześniej nierozstrzygnięte. Do badania zadań trudniejszych niż opisane przykłady często potrzebne są bardziej subtelne własności topologiczne kompleksów $\mathcal{I}$ i $\mathcal{O}$ niż tylko spójność. Przydaje się, na przykład, lemat Spernera oraz różne metody wykrywania dziur w przestrzeniach: grupy homotopii, w szczególności grupa podstawowa i grupy homologii, które dla kompleksów symplicjalnych nietrudno obliczyć. Możliwe, że topologia ma jeszcze dużo do powiedzenia w informatyce...