ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1994

Seria: INFORMATYKA z. 25 Nr kol. 1236

Rafał FAGAS

OBIEKTOWA STRUKTURA DANYCH EMULUJĄCA KOMPUTER STEROWANY PRZEPŁYWEM DANYCH I JEJ WYKORZYSTANIE W BUDOWIE INTERAKCYJNYCH SYSTEMÓW STEROWANIA, SYMULACJI I MODELOWANIA

Streszczenie. Artykuł jest prezentacją struktury obiektów, tworzącej emulator komputera sterowanego przepływem danych, uwzględniającą w sposób szczególny zastosowania takiego emulatora w interakcyjnych systemach sterowania, symulacji i modelowania. Przedstawiono w nim hierarchię klas, algorytmy przetwarzania oraz potencjalne zastosowania emulatora[1].

OBJECT ORIENTED STRUCTURE EMULATING DATAFLOW COMPUTER AND ITS APPLICATION IN INTERACTIVE SIMULATION AND MODELLING SYSTEMS

Summary. This article presents object-oriented structure of dataflow computer emulator and its applications in interactive simulation systems. Class hierarchy, algorithm and potential applications of this emulator are also exposed.

[1] Pracę wykonano w ramach grantu KBN 3 P406 011 04

DIE OBJEKTORIENTIERTE DATENSTRUKTUR ZUR EMULATION EINES DATENFLUß-COMPUTERS UND IHRE AUSNUTZUNG IN DEN INTERAKTIVEN STEUERUNGS-, SIMULATIONS- UND MODELLIERUNGSSYSTEMEN

Zusammenfassung. Im Artikel wurde die objektorientierte Struktur eined Datenfluß-Computer-Emulators dargestellt, die insbesondere seine Ausnutzung in den interaktiven Steuerungs-, Simulations- und Modellierungssystemen berücksichtigt. Die Klassenhierarchie, die Verarbeitungsalgorithmen und potentielle Verwendungsbereiche des Emulators wurden dargestellt.

1. Wstęp

Systemy sterowania i symulacji wciąż jeszcze kojarzą się nam ze specjalistycznymi, trudnymi do opanowania językami opisów symulacji, z mało atrakcyjną, uproszczoną szatą graficzną, z koniecznością żmudnego uczenia się nowego systemu od podstaw. Tymczasem masowe wejście na rynek systemów komunikacji wizualnej (MS-Windows czy choćby X-Windows) i ustanowienie przez nie pewnych standardów tejże komunikacji, odmieniło w sposób diametralny filozofię budowania i testowania modeli symulacyjnych. Już nie żmudne wpisywanie tasiemcowych programów symulacji, ale interakcyjnie budowane schematy blokowe są w chwili obecnej szczytem mody. Znakomitą większość programów służących symulacji i sterowaniu charakteryzuje obecnie filozofia kontaktu z użytkownikiem oparta na jego dotychczasowych doświadczeniach. Programów tych nie trzeba się długo "uczyć". Ikony, rozwijane menu, podręczna pomoc, a nade wszystko model w łatwo przyswajalnej dla nie-informatyka postaci schematu blokowego sprawiają, że posługiwanie się nimi nie tylko nie wymaga długich studiów, ale niesie ze sobą również pewną dozę swoistej przyjemności.

Ceną za prostotę obsługi jest jednakże niepomiernie większy wysiłek włożony przez programistów w opracowanie programów służących symulacji i sterowaniu. Rośnie złożoność struktur danych, a moduły komunikacji wizualnej z użytkownikiem zaczynają stanowić dominującą część kodu programu. W tej sytuacji podstawowym problemem staje się zaprojektowanie takich struktur danych do przechowywania programów symulacji i sterowania, aby były one podatne na zmiany wprowadzane przez użytkownika i to na zmiany wprowadzane za pomocą urządzeń wskazujących, takich jak myszka lub pióro świetlne. Struktury takie musiałyby wykazywać się dużą elastycznością łączenia (jako że program będzie miał postać schematu blokowego), a rozszerzanie repertuaru dostępnych funkcji (lub

podprogramów dostępu do specyficznych urządzeń sterowania) powinno być maksymalnie uproszczone, ze względu na konieczną dla utrzymania programu na rynku, przyszłą rozbudowę.

Spróbujmy zaprojektować strukturę obiektów służących do przechowywania programów symulacji i sterowania. Już na pierwszy rzut oka widać, że najbardziej naturalne dla tego problemu byłoby podejście obiektowe. Wszak każdy element schematu blokowego idealnie odpowiada obiektowi w rozumieniu obiektowych technik programowania. Mamy tu pole do wykorzystania zarówno dziedziczenia, jako że elementy schematu spełniają bardzo podobne funkcje, jak i polimorfizmu, bo pomimo podobieństw, działają jednak w różny sposób. W dodatku każdy element jest swego rodzaju "czarną skrzynką", z wyodrębnionymi wejściami i wyjściami, a więc spełniony mamy obiektowy postulat "kapsułkowania" danych.

2. Podstawowe pojęcia

W poniższych rozważaniach będziemy podchodzić do problemu symulacji na pewnym, dość wysokim, poziomie abstrakcji. Symulacja zadanego procesu odbywać się będzie w pewnym skończonym czasie, zwanym tutaj *czasem symulacji*. Czas ten dzielić będziemy na krótkie, dyskretne odcinki — *kroki symulacji*. Stan procesu określać będziemy z dokładnością do tychże kroków symulacji. Symulowany proces przebiegać będzie w pewnym układzie połączonych ze sobą elementów przetwarzających, z których każdy posiadać może dowolną ilość *wejść* i *wyjść*, poprzez które wysyła bądź otrzymuje od innych elementów *sygnały*. Układ taki wraz z procesem symulacji nazywać będziemy *modelem symulacyjnym*.

3. Pierwsze podejście

Zgodnie z uprzednimi definicjami, model symulacyjny, maksymalnie uogólniony, jawi się nam jako pewien zbiór *elementów* przetwarzających połączonych ze sobą drogami, po których przebiegają *dane*. Dane te wchodzą do wnętrza elementu poprzez pewne *wejścia*, ulegają przetwarzaniu, w którego efekcie inne dane pojawiają się na *wyjściach* elementu. Jak widać, jest to w istocie programowy odpowiednik komputera sterowanego przepływem danych, jako że to właśnie napływ danych na wejścia elementu inicjuje przetwarzanie. Wystarczy zatem stworzyć taki emulator komputera sterowanego przepływem danych, aby uzyskać idealne narzędzie do tworzenia łatwo modyfikowalnych modeli.

4. Implementacja, czyli drugie podejście

Wymyślona przez nas idealna struktura w postaci komputera sterowanego przepływem danych, oglądana przez pryzmat implementacji, nie wygląda już tak różowo. Maszyny sterowane przepływem danych z założenia składają się z wielu procesorów, tak więc równoległe przetwarzanie jest dla nich rzeczą naturalną. Maszyny jednoprocesorowe, dominujące na rynku, swoją pseudorównoległość okupują stratami czasowymi na tzw. przełączanie kontekstu.

Zatem uruchomienie zbyt wielu zadań na maszynie jednoprocesorowej spowoduje, że program będzie wykonywał się o wiele wolniej niż gdybyśmy całkowicie zrezygnowali ze współbieżności. W naszym przypadku liczba równoległych procesów byłaby równa ilości elementów przetwarzających, a więc z założenia (bo nie chcemy ograniczać użytkownika) bardzo duża. Z tego też powodu, nie bez żalu zmuszeni jesteśmy zrezygnować ze współbieżności elementów struktury. Paradoksalnie zyskujemy w ten sposób na czasie, w jakim struktura będzie realizować swoje obliczenia.

Tym samym jednak zrezygnowaliśmy z tzw. czasu rzeczywistego. W wariancie współbieżnym elementy działały w czasie zbliżonym do czasu rzeczywistego, a w dodatku asynchronicznie względem siebie, co dosyć dobrze oddawało istotę symulacji. Skoro jako się rzekło, w naszym przypadku współbieżność nie wchodzi w grę, musimy w jakiś sposób zastąpić ten naturalny sposób prowadzenia symulacji w czasie rzeczywistym. Stwórzmy więc substytut czasu rzeczywistego w postaci *czasu symulacji*.

Załóżmy zatem, że elementy struktury będą działały w sposób współbieżny, ale nie w tym zwyczajnym czasie, na który nie mamy żadnego wpływu i który wymyka się spod wszelkiej kontroli, ale w naszym, stworzonym przez nas i przez nas sterowanym czasie symulacji. Ponieważ to my kontrolujemy czas symulacji, możemy sprawić, że całe przetwarzanie odbędzie się w jednym jego kwancie. Po prostu w naszym modelu czas będzie się posuwał do przodu tylko wtedy, gdy cała struktura wykona jeden krok przetwarzania.

Zauważmy, że nie zrywamy w ten sposób całkowicie z czasem rzeczywistym. Przy odpowiednio szybkim komputerze i zastosowaniu efektywnych algorytmów, czas trwania kroku symulacji mierzony czasem symulacji i czasem rzeczywistym mogą być do siebie bardzo zbliżone.

5. Obiekty tworzące strukturę

Abstrahując od szczegółów, widzimy, że niezbędne będzie wprowadzenie co najmniej trzech klas obiektów:

- klasy obiektów przenoszących *dane* (odpowiednik znaczników danych w komputerach przepływowych),
- klasy obiektów wykonujących *przetwarzanie*,
- klasy obiektów sterujących *czasem symulacji*.

5.1. Obiekty przenoszące dane

Naszym modelem sterować będą dane. Dane te będą przenosić najróżniejsze informacje, a więc nie ma sensu sztywne określanie ich struktury. Danymi mogłyby być impulsy zegara, wartości logiczne, wartości analogowe (temperatura, ciśnienie etc.), a nawet struktury danych (rekordy, listy, kolejki). Problem w tym, aby maksymalnie uprościć posługiwanie się nimi. Na szczęście programując w języku zorientowanym obiektowo, możemy skorzystać z mechanizmu kapsułkowania danych. Dzięki niemu oraz dzięki mechanizmom *dziedziczenia i polimorfizmu* uzyskujemy możliwość operowania na wielu typach danych zamkniętych (zakapsułkowanych) w obiektach.

Efekt ten osiągamy deklarując abstrakcyjną klasę o nazwie dajmy na to **DataPack**. Klasa ta nie będzie przenosić jeszcze żadnych danych. Będzie ona stanowić szkielet dla innych klas i definiować standardowy sposób posługiwania się obiektami danych. Jedyną informacją jaką przenosić będą obiekty klasy **DataPack**, jest typ danych, potrzebny w przyszłości do sprawdzania zgodności typów podczas operacji na danych.

Jeżeli chodzi o standardowe metody, które powinny zostać zdefiniowane dla wszystkich klas danych (dziedziczących po **DataPack**), to obejmują one metody:

- zerowania danych,
- pobierania typu danych,
- tworzenia kopii danych,
- sprawdzania równości danych (definicja operatora porównania).

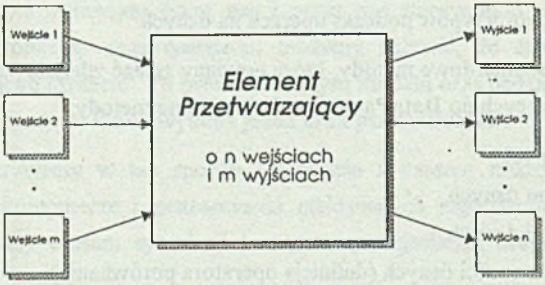
Metody te są wystarczające dla zbudowania naszej struktury i zapewnienia możliwości operowania w jednolity sposób danymi różnych typów. Zdefiniowaliśmy zatem podstawową "paczkę danych" służącą do pobudzania naszej sieci elementów przetwarzających oraz przenoszącą rezultaty obliczeń.

5.2. Obiekty elementów przetwarzających

Elementy przetwarzające mogą być różne. Sumatory, elementy całkujące, różniczkujące, elementy przetwarzające za pomocą zadanej funkcji etc. Co więcej, jako elementy w sieci przetwarzającej mogłyby znaleźć miejsce także elementy wizualizujące efekty obliczeń, elementy wykonujące operacje sterowania za pomocą urządzeń wejścia/wyjścia, elementy archiwizujące wyniki oraz wszelkie inne, jakie tylko twórca programu zdoła sobie wyobrazić. Cechą wspólną wszystkich tych elementów jest to, że posiadają wejścia, na które napływają dane, oraz wyjścia, przez które wypływają wyniki obliczeń. Ponieważ chcemy objąć w swoim projekcie jak największą ilość możliwych elementów, podobnie jak to uczyniliśmy w przypadku projektowania danych, powinniśmy zdefiniować abstrakcyjną klasę bazową, implementującą zachowanie wspólne dla wszystkich możliwych elementów.

Niech klasa ta nazywa się **ElementBase**. Każdy obiekt tej klasy posiadać będzie pewną liczbę wejść oraz pewną liczbę wyjść. Wejścia i wyjścia mogą być różnego typu (analogowe, cyfrowe itp.). Pojawienie się nowych danych na wszystkich wejściach danego elementu (czyli skompletowanie znaczników danych) powinno spowodować przetworzenie tych danych i przekazanie wyników w odpowiedniej postaci na odpowiednie wyjścia. Wejścia i wyjścia elementów mogą być ze sobą połączone, a więc wysłanie danej na wyjście jednego elementu spowoduje automatyczne wysłanie tej danej na wszystkie wejścia z nim połączone, czego następstwem będzie swego rodzaju łańcuchowa reakcja przetwarzania.

Mówimy tu cały czas o wejściach i wyjściach. Wejścia i wyjścia również będą obiektami. Cały element przetwarzający o n wejściach i m wyjściach został przedstawiony graficznie na rys. 1 (strzałki obrazują przepływ danych).



Rys. 1. Element przetwarzający, jego wejścia i wyjścia
Fig. 1. Processing element, its inputs and outputs

Obiekty wejść i wyjść wykonują stosunkowo proste funkcje. Ich działanie ogranicza się do przyjmowania lub wysyłania danych oraz do informowania o swoim stanie. Szczególnie ważny jest stan wejść, jako że warunkuje on przetwarzanie w elemencie.

Wejście może przybierać jeden z trzech stanów:

- **Wejście z nowymi danymi.** Wejście wchodzi w ten stan z chwilą nadejścia nowych danych.
- **Wejście obsłużone.** Wejście wchodzi w ten stan po odebraniu danych przez element przetwarzający.
- **Wejście w stanie początkowym.** Wejście znajduje się w tym stanie na początku pierwszego kroku przetwarzania. Wprowadzenie tego stanu jest konieczne, ażeby umożliwić sprzężenia zwrotne w strukturze elementów.

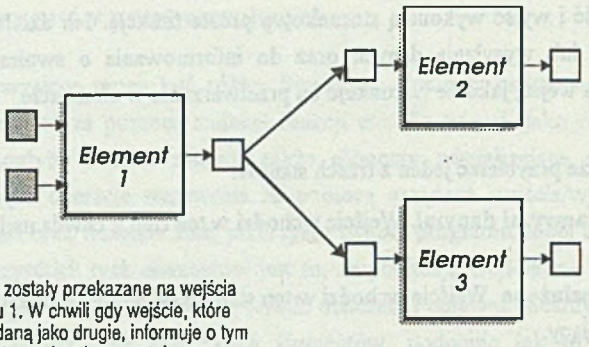
Wejścia wykonują jeszcze jedno zadanie, kluczowe z punktu widzenia przetwarzania. Za każdym razem, kiedy na wejściu pojawią się nowe dane, wejście informuje o tym element, z którym jest związany. Element reaguje na to sprawdzeniem, czy wszystkie wejścia są gotowe (tzn. czy na wszystkich wejściach znajdują się nowe dane lub stany początkowe)[2] i jeżeli tak, wykonuje przetwarzanie i wysyła jego wyniki na wyjścia.

Wyjścia dystrybuują otrzymane od elementu macierzystego dane do wszystkich wejść, z którymi są połączone, a trzeba tu wspomnieć, że jedno wyjście może być podłączone do dowolnej ilości wejść.

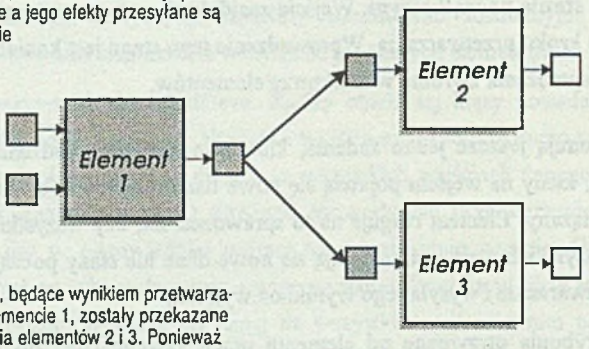
Wydawać by się mogło, że w zaistniałej sytuacji właściwie nic więcej nie jest nam potrzebne. Mamy paczki danych, mogące przenosić dowolne typy danych, mamy elementy przetwarzające, również na tyle ogólne, że możemy zdefiniować dowolne przetwarzanie i mamy wreszcie wejścia i wyjścia na tyle "inteligentne", że same wiedzą, co zawierają i potrafią o tym zawiadomić element macierzysty.

Więc może wystarczy tylko utworzyć sieć połączeń między wejściami i wyjściami elementów i posłać na wejścia tak powstałego modelu dane wejściowe? Przecież reakcja łańcuchowa, wywołana pojawieniem się tych danych na wyjściach, powinna zaowocować wynikami przetwarzania na wyjściach modelu, tak jak na rys.2.

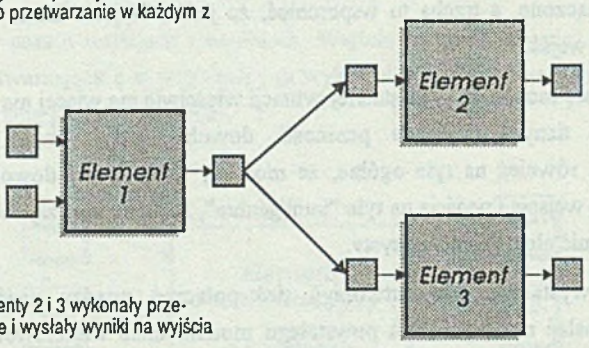
[2] Stan początkowy jest równoważny gotowości wejścia. Założenie to jest konieczne dla struktur ze sprzężeniami zwrotnymi.



a) Dane zostały przekazane na wejścia elementu 1. W chwili gdy wejście, które dostało daną jako drugie, informuje o tym element 1, uruchomione zostaje przetwarzanie a jego efekty przesyłane są na wyjście



b) Dane, będące wynikiem przetwarzania w elemencie 1, zostały przekazane na wejścia elementów 2 i 3. Ponieważ są to jedyne wejścia tych elementów, inicjuje to przetwarzanie w każdym z nich.



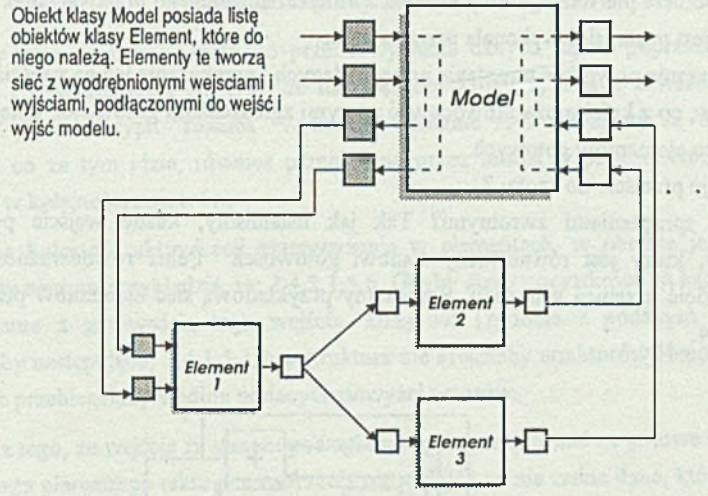
c) Elementy 2 i 3 wykonały przetwarzanie i wysłały wyniki na wyjścia modelu.

Rys. 2. Przepływ danych w przykładowej strukturze
Fig. 2. Data flow in example structure

Niestety, jest tak tylko w teorii. W praktyce bowiem przekazywanie danych pomiędzy elementami struktury odbywa się poprzez wywołania funkcji. Na początku wywoływana jest funkcja przyjęcia danych (w kontekście odpowiedniego obiektu). Funkcja ta, po sprawdzeniu gotowości wszystkich wejść, wykonuje przetwarzanie i wysyła wyniki, tj. wywołuje funkcje

przyjęcia danych, kolejno we wszystkich elementach, do których ma je przekazać z uwagi na strukturę połączeń. Funkcje przyjęcia danych w tych elementach powtarzają cały ten proces. Tak więc im głębiej przetwarzanie "zanurza się" w strukturę, tym większą mamy ilość rekurencyjnie wywołanych funkcji. Taka głęboka rekurencja z kolei doprowadzałaby do częstych zawieszeń systemu na tle przepełnienia stosu. Ponadto każde sprzężenie zwrotne w strukturze powodowałoby nieskończoną rekurencję!

Zachodzi więc potrzeba zbudowania klasy obiektów kontrolujących przebieg symulacji w modelu.



Rys. 3. Wzajemne zależności obiektów struktury
Fig. 3. Relationships between objects

5.3. Obiekty kontroli symulacji

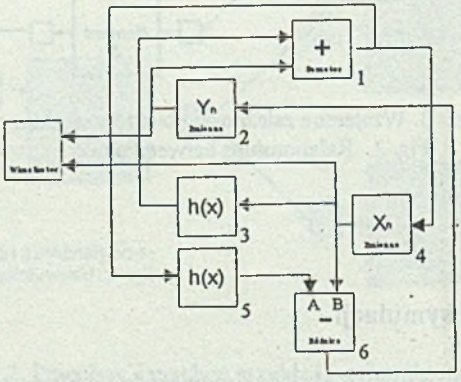
Stwórzmy klasę obiektów o nazwie **Model**. Każdy obiekt tej klasy sprawowałby pieczę nad powierzoną sobie siecią elementów, z wyodrębnionymi wejściami i wyjściami, które można by nazwać *wejściami modelu* i *wyjściami modelu*. Zadaniem obiektu byłoby wykonanie na powierzonej sobie sieci jednego kroku symulacji, tj. wykonanie przetwarzania na otrzymanych danych i skierowanie wyników na wyjścia modelu. Ponieważ nie można wykonywać przetwarzania w sieci elementów w sposób rekurencyjny, spróbujmy określić algorytm przetwarzania.

5.3.1. Algorytm przetwarzania

Niech elementy przetwarzające, zamiast od razu po otrzymaniu kompletu danych wykonywać przetwarzanie, wykonują tylko *przygotowanie do przetwarzania*. Przygotowanie do przetwarzania polega na poinformowaniu modelu macierzystego[3] o gotowości elementu. W odpowiedzi model wstawia dany element do kolejki elementów gotowych. Zatem algorytm wykonania jednego kroku symulacji wyglądałby następująco:

1. Dane z wejść modelu są rozsyłane na odpowiednie wejścia sieci.
2. Te elementy, które po otrzymaniu danych są gotowe, informują o tym model, który te informacje kolejkuje.
3. Model pobiera pierwszy gotowy element z kolejki i inicjuje jego przetwarzanie. Jeżeli kolejka jest pusta, sieć wykonała przetwarzanie.
4. Przetwarzanie powoduje powstanie nowych danych i przekazanie ich na wejścia innych obiektów, co z kolei może zaowocować nowymi zgłoszeniami gotowości, kolejkowanymi w kolejce elementów gotowych.
5. Następuje przejście do kroku 3.

A co ze sprzężeniami zwrotnymi? Tak jak ustaliliśmy, każde wejście posiada stan początkowy, który jest równoważny stanowi gotowości. Pełna równoważność mogłaby jednak zakłócić przebieg symulacji. Rozważmy przykładową sieć elementów przedstawioną na rysunku 4.



Rys. 4. Przykładowa sieć elementów
Fig. 4. Example of elements connections

[3] Obiektu klasy Model, do którego należy dany element

Sieć ta wykreśla za pomocą elementu **Wizualizator** tzw. atraktory Henona, powstające z punktów o współrzędnych wyznaczonych ze wzorów:

$$x(n+1) = y(n) + h(x(n))$$
$$y(n+1) = -x(n) + h(x(n+1))$$

gdzie:

$$h(x) = A * x + \frac{(2*(1-A)*x^2)}{1+x^2}$$

A-parametr
n-numer kroku symulacji

Elementy typu "zmienna" służą do przechowywania danych (np. z poprzednich kroków symulacji) i charakteryzują się tym, że na początku kroku symulacji zawsze są w stanie gotowości. Element typu "różnica" wykonuje działanie A-B. Elementy są dodawane do modelu, a co za tym idzie, również przeglądane przez model w poszukiwaniu elementów gotowych w kolejności numerów.

Poprawna kolejność aktywizacji przetwarzania w elementach, w obrębie jednego kroku symulacji w naszym przykładzie, to: 2,4,3,1,5,6. Gdyby stany początkowe na każdym wejściu były tożsame z gotowością tego wejścia, kolejność (zgodnie z podanym algorytmem) wyglądałaby następująco: 2,4,1,5,3,6, a struktura nie kreśliłaby atraktorów Henona, ponieważ iteracja nie przebiegałaby według podanych powyżej wzorów.

Wynika z tego, że wejście ze stanem początkowym możemy uznać za gotowe tylko jeżeli w czasie całego pierwszego taktu przetwarzania nie nadejdą na nie żadne dane, które spowodują jego przejście w stan gotowości. Elementy, których choć jedno wejście jest w stanie początkowym, powinny zatem zostać zaktywizowane po elementach z wszystkimi wejściami w stanie gotowości, czyli dopiero wtedy, gdy nie będzie już szansy na to, że ich aktywizacja nastąpi poprzez ktoreś ze sprzężeń zwrotnych. Innymi słowy, stan początkowy na jakimś wejściu należy potraktować podobnie jakby wejście było w stanie gotowości, tj. wstawić element do kolejki. Należy tu jednak wprowadzić inną kolejkę, ponieważ będziemy pobierali z niej elementy dopiero wówczas, gdy skończą się elementy w kolejce podstawowej. Zmodyfikowany algorytm wyglądać będzie następująco:

1. Dane z wejść modelu są wysyłane na odpowiednie wejścia sieci.
2. Te obiekty, które po otrzymaniu danych są gotowe, informują o tym model, który przechowuje te informacje w dwóch kolejkach, *główna* przechowuje informacje o elementach, w których każde wejście jest gotowe i żadne nie jest w stanie początkowym,

pomocnicza o elementach, w których wszystkie wejścia są bądź gotowe bądź w stanie początkowym, ale co najmniej jedno jest w stanie początkowym.

3. Model pobiera pierwszy gotowy element z kolejki głównej i inicjuje jego przetwarzanie. Jeżeli kolejka jest pusta, model przechodzi do kroku 4. Jeżeli nie, to do kroku 5.
4. Pobieramy element z kolejki pomocniczej, sprawdziwszy, czy nie został on już w danym kroku wykonany. Istnieje bowiem możliwość, że element przesłany do kolejki pomocniczej, na skutek późniejszego uzyskania kompletu wejść, znalazł się też w kolejce głównej i wykonał już przetwarzanie. Model inicjuje przetwarzanie w pobranym elemencie. Jeżeli kolejka pomocnicza jest pusta, to wszystkie elementy wykonały przetwarzanie.
5. Przetwarzanie powoduje powstanie nowych danych i przekazanie ich na wejścia innych elementów, co z kolei może zaowocować nowymi zgłoszeniami gotowości, kolejkowanymi w kolejce głównej.
6. Następuje przejście do kroku 3.

W ten sposób zdefiniowaliśmy algorytm przetwarzania w tak zbudowanej sieci elementów. Algorytm powyższy nie powoduje głębokich rekurencji, można go zatem stosować nawet dla bardzo rozbudowanych sieci elementów. Liczba elementów w kolejkach jest stabilna i zależy w głównej mierze od tego, ile wejść podłączymy do jednego wyjścia oraz od ogólnej konfiguracji sieci. To oraz fakt, że pamięć dla nowych elementów kolejek może być przydzielana i zwalniana dynamicznie sprawia, że przy obecnie "obowiązujących" rozmiarach pamięci operacyjnej nie powinno jej emulatorowi brakować.

6. A co z czasem?

Na początku wprowadziliśmy pojęcie *czasu symulacji*. Pojęcie to jednak nie pojawiło się już później. Najwyższy czas, aby do niego powrócić.

Powinniśmy wprowadzić do naszych rozważań obiekt odgrywający rolę zegara symulacji. Sprowadźmy czas do poziomu danej generowanej cyklicznie przez ten obiekt na podstawie wartości uprzednio zadanych, to jest: początkowej wartości czasu, wartości końcowej i kroku, o który zwiększać będziemy wartość zegara w każdym takcie. W takim przypadku podłączenie zegara do naszego modelu jest już bardzo proste. Wystarczy dane generowane przez zegar doprowadzić do jednego z wejść modelu i zapewnić możliwość poinformowania zegara o tym, że model wykonał już takt symulacji i że zegar może wygenerować kolejny takt.

7. Potencjalne możliwości struktury

Zaprojektowaliśmy strukturę wzorowaną na komputerze sterowanym przepływem danych. Pomimo że projektowaliśmy ją z myślą o wykorzystaniu w programach symulacji i sterowania, jej potencjalne zastosowania są o wiele szersze. Obejmują one całą klasę problemów, w których zachodzi potrzeba interakcyjnego modyfikowania przebiegu obliczeń bądź ogólnie rzecz biorąc, przebiegu jakiegokolwiek przetwarzania. Struktura ta może służyć jako podstawa dla programów obliczeniowych, dla programów służących do planowania (*ang. Project managing*), a nawet dla generatorów aplikacji. Wszystko zależy od odpowiedniego zdefiniowania podstawowych elementów przetwarzających. W zasięgu tej struktury leży również możliwość definiowania nowych elementów przez użytkownika poprzez składanie ich z elementów podstawowych, co umożliwiałoby użytkownikowi tworzenie bibliotek własnych "podprogramów przetwarzania". Poza tym, oczywiście, struktura jest przystosowana do edycji za pomocą urządzeń wskazujących, co umożliwia wizualne projektowanie sieci połączeń.

Recenzent: Dr hab. inż. Stanisław Wołek

Wpłynęło do Redakcji 3 sierpnia 1993r.

Abstract

This article presents object-oriented structure of simplified dataflow computer emulator and its applications in interactive simulation systems. Fundamental definitions concerning simulation problems are presented in section 2. Section 3 presents first, simplified approach to the problem. Implementation issues are discussed in section 4. Section 5 presents proposed class hierarchy in our emulator, consisting of abstract base classes for data bearing objects, processing element objects, input objects, output objects and model objects. Subsection 5.3.1 defines first version of processing algorithm in model object. In subsequent subsections, this algorithm is modified up to its final version. Last section presents possible (other than simulation) applications for designed emulator.