STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Marcin DĄBKIEWICZ, Jarosław KOSZELA
Wojskowa Akademia Techniczna, Wydział Cybernetyki

ZASTOSOWANIE KONSTRUKCJI DEKLARATYWNYCH DO ROZPRASZANIA I OPTYMALIZACJI PRZETWARZANIA DANYCH[1]

Streszczenie. W artykule zostanie przedstawiona koncepcja rozszerzenia własności imperatywnych języków programowania o elementy konstrukcji deklaratywnych, które pozwalają na oszacowanie kosztów przetwarzania wraz z możliwością określenia niezależnych kroków przetwarzania, które mogą podlegać rozproszeniu zarówno w obrębie danego węzła przetwarzania (np. GPU lub różne rdzenie CPU), jak i całego systemu rozproszonego, biorąc pod uwagę efektywność całego procesu przetwarzania (optymalizacja).

Słowa kluczowe: języki programowania, rozproszenie i optymalizacja przetwarzania

APPLYING DECLARATIVE CONSTRUCTIONS FOR DISTRIBUTING AND OPTIMIZING DATA PROCESSING

Summary. In that article will be shown concept of extending imperative properties of programming languages with declarative constructions. Those constructions allow estimate the cost of processing with the ability to declare independent processing steps. Steps can be then dispersed within the processing node (i.e. GPU or different cores of CPU) as well as whole distributed system. Effectiveness of whole computation process (optimization) is taken into consideration during that distribution.

Keywords: programming languages, distributing and optimizing of data processing

[1] Finansowanie ze środków NCBiR – umowa 0001/R/ID3/2011/01.

1. Wstęp

Za obecnym, dynamicznym rozwojem sprzętu komputerowego nie idzie niestety taki sam progres metod, technik i narzędzi programowych wytwarzania systemów informatycznych. Systemy wieloprocesowe czy rozproszone, niedawno obecne tylko w specjalizowanych laboratoriach czy centrach przetwarzania rozwiązania, obecnie dostępne są niemal w każdym domowym komputerze, laptopie czy też w telefonie komórkowym (wielordzeniowe CPU, GPU, duże zasoby pamięciowe itp.). Wykorzystanie tak dużych i powszechnie dostępnych mocy obliczeniowych przy typowych aplikacjach czy systemach biznesowych jest praktycznie znikome, jeśli nie liczyć specjalizowanych i przygotowanych narzędzi i rozwiązań, jakimi są np. silniki bazodanowe, serwery aplikacji i webowe czy systemy operacyjne. Problemy zaczynają się już na etapie procesu wytwórczego, gdzie przy obecnie stosowanych metodach, technikach i narzędziach należy odpowiednio zdefiniować elementy logiczne i konstrukcyjne wytwarzanego systemu w taki sposób, aby mogły wykorzystać efektywniej dostępne zasoby (np. wyodrębnienie elementów systemów mogących działać współbieżnie i/lub równolegle). Podejście to zwiększa koszty wykonania systemu i wymaga od wykonawców większej wiedzy i doświadczenia w budowaniu takich systemów. Proponowane w artykule rozwiązanie polega na rozszerzeniu obecnie wykorzystywanych języków imperatywnych o elementy deklaratywne i ich unifikację z elementami imperatywnymi w celu rozszerzenia możliwości, jakie dają konstrukcje deklaratywne. Zastosowanie tego rodzaju rozwiązań powinno usprawnić i zwiększyć efektywność wykorzystania dostępnych zasobów systemu, w szczególności przez rozproszenie oraz optymalizację przetwarzania danych.

2. Języki przetwarzania danych – opis problemu

2.1. Języki programowania

Obecnie powszechnie wykorzystywane do budowy systemów informatycznych języki programowania w większości są językami imperatywnymi (np. C# .NET, JAVA itp.). Języków deklaratywnych, choć nie jest to liczna grupa języków, jest niewiele (np. SQL, Prolog, Datalog) np. w stosunku do grupy języków imperatywnych, mimo iż ich potencjał w kategoriach możliwości rozpraszania i optymalizacji jest bardzo duży. W najczęstszych zastosowaniach przy budowie systemów informatycznych używane są zasoby, które w zależności od poziomu dekompozycji można wyodrębnić, np. systemy operacyjne, infrastruktura sieciowa czy bazy danych lub, bardziej szczegółowo: pamięć, procesory, protokoły i sieć komputerowa [1]. Języki deklaratywne są w stanie efektywniej wykorzystać te zasoby bez konieczności ponoszenia dużego wysiłku i dużego kosztu w całym procesie wytwórczym oprogramowania.

Można się o tym przekonać, obserwując chociażby funkcjonowanie mocno obciążonych serwerów bazodanowych. Języki imperatywne nie mają niestety takiej siły, bynajmniej nie da się jej osiągnąć w tak prosty sposób.

Języki imperatywne to języki, w których program rozumiany jest jako ciąg instrukcji wykonywanych sekwencyjnie, które pozwalają programiście zarówno na określenie celu, jak i sposobu dojścia do niego. Ze względu na to, iż jest to niski poziom logiczny, interpreter nie ma dużego pola manewru, jeśli chodzi o sposób i optymalizację wykonania lub rozproszenia operacji. Zresztą sam sposób pisania programu – nienastawiony na współbieżność – utrudnia wykorzystanie całego potencjału środowiska. Aby napisać efektywny program, programista nie uniknie tematu wątków oraz ich synchronizacji. Z kolei nawet biegłość w temacie współbieżności nie spowoduje pełnego wykorzystania możliwości. Ten sam blok programu w zależności od parametrów wejściowych należałoby wykonywać w inny sposób. Nie byłoby przecież sensu rozdzielać wykonania np. wyliczeń w pętli na kolekcji 5-elementowej, z kolei dla 5 mln elementów równoległość jest jak najbardziej wskazana. Dynamikę wykonań dają nam za to języki deklaratywne.

Języki deklaratywne to języki, w których za pomocą poleceń określamy cel, który chcemy osiągnąć, a sposobem wykonania zadania i wykorzystania zasobów zajmuje się środowisko wykonawcze. Przed wykonaniem operacji wyznaczanych jest wiele planów wykonania. Dla każdego z nich wyliczana jest złożoność obliczeniowa, a następnie wybierany jest plan optymalny przy pewnej ustalonej funkcji celu. Przy wyznaczaniu planów wykonania i wyliczaniu złożoności bierze się pod uwagę również koszt oraz wolumen danych, w tym jego statystyki. Może się więc okazać, że pewny plan wykonania jest nieefektywny ze względu na zbyt małą/dużą liczbę elementów, na których przeprowadzana będzie operacja. Efekt jest taki, że za każdym razem sposób wykonania może być inny, liczy się tylko wynik końcowy.

Zaletą deklaratywów jest to, iż są one logicznie zamknięte i ukierunkowane na cel, a nie na realizację. Założeniem ich konstrukcji jest jak najprostszy sposób wyrażenia celu przy pominięciu sposobu jego realizacji lub, za prof. K. Subietą: ich celem jest *minimalizacja elementów w wyrażeniach języka, które odnoszą się do środowiska komputerowego, a skupienie się na maksymalizacji elementów, które odnoszą się do dziedziny przedmiotowej i modelowania pojęciowego tej dziedziny*[2].

Widać tu pewną „dziurę”. Języki deklaratywne są zbyt ogólne i ograniczone, aby budować złożone systemy. Główną ich wadą jest to, iż nie dają odpowiedniego poziomu ekspresyjności (np. kompletności obliczeniowej) [9]. Jednak w językach imperatywnych, pomimo posiadania tej cechy, występuje zbyt duża szczegółowość i rozdrobnienie, przy tym są one mało elastyczne. Elastyczność rozumiana jest tutaj jako umiejętność dopasowania się zamo-

[2] K. Subieta, T. Wardziak: Języki i środowiska programowania baz danych. PJWSTK, Warszawa 2006, http://edu.pjwstk.edu.pl/wyklady/jps/scb/wyklad13/section1.html

delowanego, skonstruowanego, zaimplementowanego oraz wykonywanego rozwiązania (skompilowanego lub zinterpretowanego) do zmieniającego się środowiska wykonawczego. Środowisko to rozumiane jest jako kompleksowa struktura techniczno-systemowa (jednostki przetwarzania, sieć, zasoby itp.).

Istnieje również grupa języków funkcyjnych. Są to filozofia i metodyka programowania, które są odmianą programowania deklaratywnego. Języki funkcyjne bazują na rachunku lambda (*λ-calculus*), opracowany,o przez Alonzo Churcha. Nacisk kładzie się tu na wartościowanie funkcji, a nie na wykonywanie poleceń. Ze względu na ograniczenia tych języków nie zastąpią one imperatywnych. Sama jednak koncepcja wyrażeń lambda została szybko przyjęta w językach imperatywnych. Jest to znaczny krok w kierunku rozpraszania i optymalizacji przetwarzania. W wielu platformach programistycznych wprowadzenie tych wyrażeń spowodowało dodanie nowych możliwości języka, takich jak uniwersalność i rozszerzalność oraz przyczyniło się do zwiększenia poprawności i czytelności kodu, a przede wszystkim podniosło poziom abstrakcji. Przykładem może być następujący listing:

```
IEnumerable<BankAccount> accounts = db.GetAccounts();

// 1. tradycyjne pobranie listy kont ze stanem ujemnym
IEnumerable<BankAccount> debt1 = new List<BankAccount>();
foreach (BankAccount account in accounts)
{
    if (account.Amount < 0)
        debt1.Add(account);
}

// 2. zapis deklaratywny z użyciem wyrażenia lambda
IEnumerable<BankAccount> debt2 = accounts.Where(x => x.Amount < 0);

// 3. zapis deklaratywny z użyciem LINQ
IEnumerable<BankAccount> debt3 = from account in accounts
                                 where account.Amount < 0
                                 select account;
```

Przedstawia on trzy sposoby pobrania wykazu kont ze stanem ujemnym. Pierwszy, tradycyjny wymaga zadeklarowania nowej kolekcji i dopisywania do niej elementów spełniających kryterium, przy czym wszelkie sprawdzenia należy wykonać samemu. Sposoby drugi i trzeci to użycie wyrażenia lambda. Określa się w nich tylko to, co chce się osiągnąć, a realizację tego zadania pozostawia się środowisku wykonawczemu. Jest to tylko znacznie uproszczony przykład, jednak im większa złożoność problemu, tym większy zysk z użycia wyrażeń lambda.

Łatwo sobie wyobrazić moc drzemiącą w tym rozwiązaniu. Dla przykładu można przyjąć założenie, że lista kont bankowych posiada 10 mln elementów, zakładając ponadto, że przy przekroczeniu 2 mln elementów w liście opłaca się zrównoleglić operację wyszukiwania i przenieść ją do kolejnego wątku, a środowisko potrafi zrobić to automatycznie, bez ingerencji programisty. Środowisko wykonawcze sprawdza uwarunkowania i decyduje się podzielić przetwarzanie na wątki (wykorzystując wszystkie dostępne rdzenie procesorów), wśród któ-

rych każdy przetwarzałby co najmniej 1 mln elementów. Przy takich uwarunkowaniach możliwe jest zrównoleglenie pracy w zakresie 1-10 dostępnych rdzeni CPU, i to bez żadnego nakładu projektanta/programisty ani zmiany oprogramowania. Niestety obecne rozwiązania nie są aż tak daleko posunięte.

2.2. Proces wytwarzania systemów

Główny nacisk na szybkość i niezawodność tworzenia aplikacji kładziony jest głównie w obszarze aplikacji biznesowych. Przełożenie wymagań na kod aplikacji, a także jego elastyczność są kluczowymi kwestiami dla biznesu. Szczególnie tyczy się to systemów, których dotyka efekt skali. Przetwarzanie dużych woluminów danych lub duża liczba wyliczeń sprawiają, że gdy danych z czasem przybywa, system staje się coraz wolniejszy.

Rozwiązaniem wskazanych problemów jest modernizacja sprzętu lub oprogramowania. Biorąc pod uwagę fakt, że:

- nie wszystkie algorytmy da się udoskonalić;
- każda modyfikacja oprogramowania niesie za sobą ryzyko jego błędnego funkcjonowania (regresja);
- czas wymagany na przejście przez wszystkie fazy developmentu jest z reguły duży;

pożądanym przez biznes rozwiązaniem jest modernizacja sprzętu. Jest to zresztą ekonomicznie uzasadnione – wielokrotnie taniej jest dokupić kość (lub kilka kości) pamięci, niż optymalizować zarządzanie pamięcią w systemie, co przecież nie pozostanie bez wpływu na wydajność rozwiązania.

Modernizacja sprzętu, pomimo że tańsza, często okazuje się bezcelowa. Dodanie kolejnych procesorów odciąży system operacyjny, ale *de facto* nie przyspieszy działania samej aplikacji, jeśli jest ona jednoprocesowa i jednowątkowa. Nawet aplikacja wielowątkowa może w pełni nie wykorzystać możliwości środowisk wieloprocesorowych lub klastrów.

Obecnie brakuje konstrukcji programistycznych, które dla zdefiniowanego programu i sprzętu, na którym będą uruchamiane, same zoptymalizują wykorzystanie zasobów. Dlatego też opracowywana koncepcja zakłada wprowadzenie rozszerzeń języków obiektowych, które rozwiązywałyby poruszane problemy. Rozszerzenia te musiałyby jednak spełniać następujące warunki:

- dotyczyłyby zarówno narzędzi, jak i środowisk obiektowych;
- posiadałyby uwspólnione typy danych;
- dawałyby odpowiedni poziom ekspresyjności języka (np. zachowywałyby kompletność obliczeniową);
- decyzje techniczne byłyby wyeliminowane z procesów modelowania i projektowania systemu lub przynajmniej zminimalizowane. Proces konstrukcji systemów zostałby przenie-

siony na poziom projektanta/programisty, przy czym byłby on zwolniony z mechanizmów synchronizacyjnych (pisanie kodu pod równoległość jest czasochłonne);
- posiadałby możliwość harmonogramowania zadań.

3. Zastosowanie konstrukcji deklaratywnych języka do efektywniejszego konstruowania systemów informatycznych

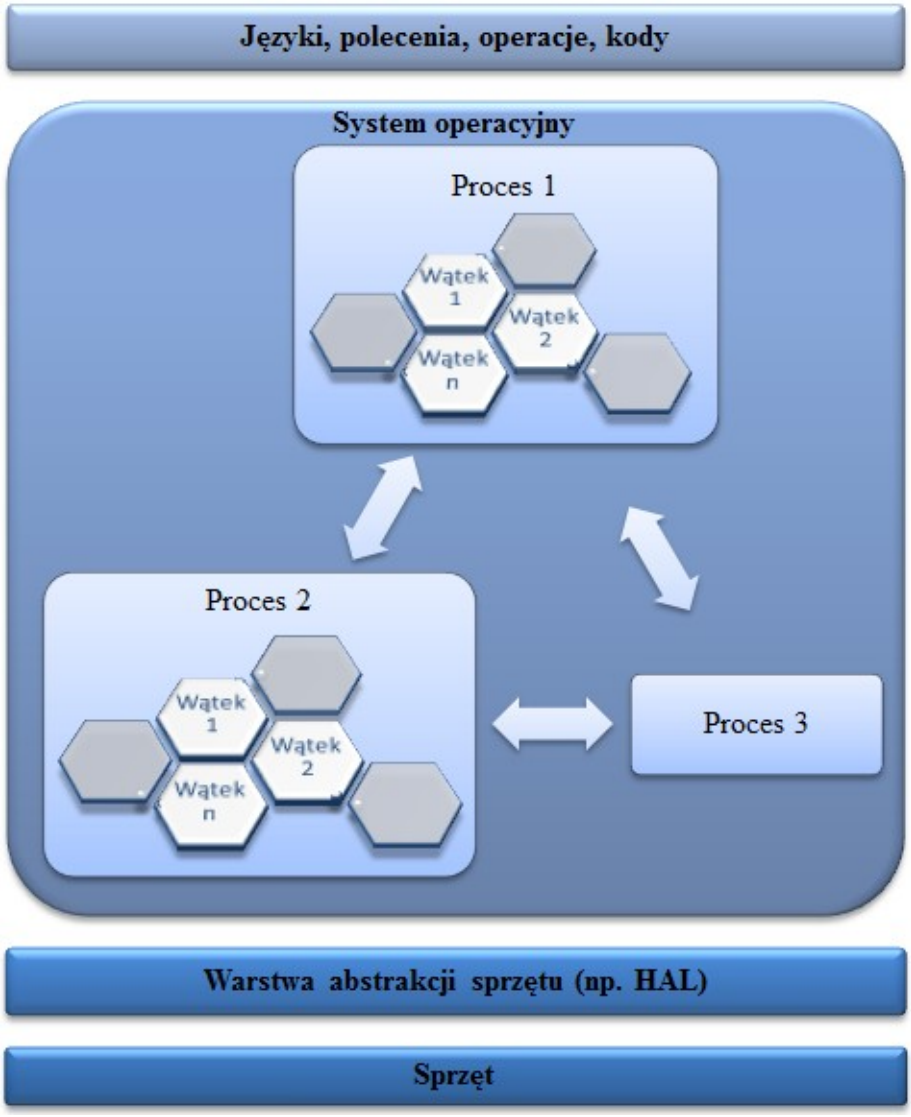
W koncepcji rozszerzenia własności imperatywnych języków programowania o elementy konstrukcji deklaratywnych, o której mowa w niniejszym artykule, skupiono się tylko na wycinku obejmującym głównie typowe aplikacje biznesowe. Pominięto tu dość obszerne zagadnienie, jakim są systemy obliczeniowe o wysokiej wydajności (ang. *High-performance computing*, w skrócie HPC). Nie jest to główny obszar zainteresowań i zastosowań rozważany przez autorów przy tworzeniu wspomnianej koncepcji. Jak już wspomniano na wstępie, skupiono się również na rozszerzaniu języków imperatywnych o deklaratywny, a nie odwrotnie. Jednocześnie istotnym aspektem jest próba unifikacji konstrukcji imperatywnych i deklaratywnych języka obiektowego w taki sposób, aby nie powstało zjawisko znane jako niezgodność impedancji języków (ang. *impedance mismatch*) [5].

Opracowywana koncepcja ma na celu wyeliminowanie silnej zależności pomiędzy etapami procesu wytwórczego (rys. 1) [6]. Celem jest stworzenie rozszerzeń języka, aby ich używanie było naturalne na etapie realizacji, jednak aby nie wymagało potrzeby ich uwzględniania na etapach modelowania i projektowania.

Rys. 1. Fragment procesu wytwórczego oprogramowania
Fig. 1. Part of a software development process

Dla przykładu budowa aplikacji rozproszonej wymusza odpowiednie modelowanie i projektowanie z uwzględnieniem konstrukcji programowych, które takowe podejście umożliwiają. Obszar zainteresowań obejmuje przy tym samą konstrukcję systemu bez uwzględniania zależności międzysystemowych oraz międzyprocesowych (rys. 2). Zależności te są obszara-

mi działań dla takich mechanizmów i abstrakcji, jak SOA, RPC, Remoting, RMI itp. i nie będą tutaj rozpatrywane ani analizowane.



Rys. 2. Fragment struktury systemu
Fig. 2. Part of a system structure

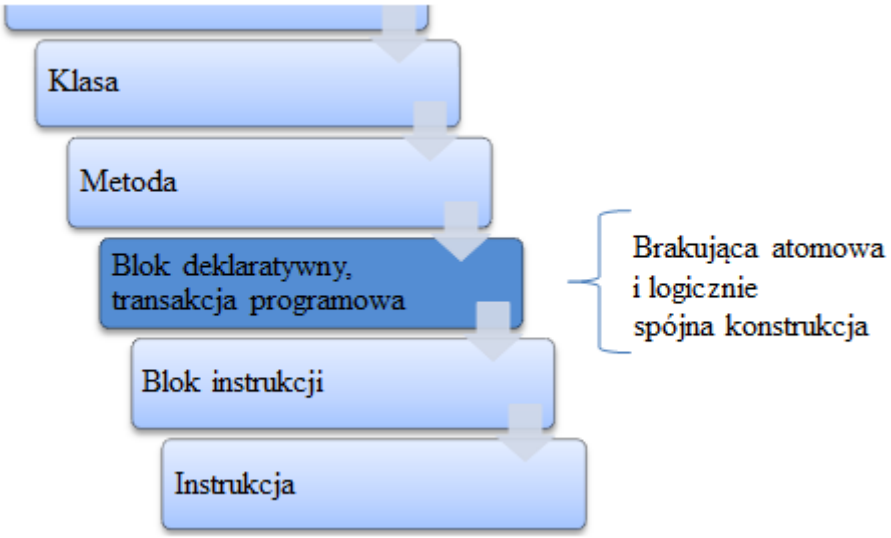
3.1. Konstrukcje językowe

Wiele współczesnych środowisk programistycznych ma już w swoim arsenale konstrukcje deklaratywne oraz bloki danych, mających specjalne znaczenie (np. blok try – catch). W niektórych z nich składnia oferuje możliwości zrównoleglania z jednoczesnym ukryciem mechanizmów synchronizacji [7]. Przykładem takiego języka jest ADA. Pomimo swych niewątpliwych zalet projektant/programista nadal zmuszony jest do myślenia w kategoriach zadań (tasków[3]) już na etapie projektowania. Z kolei użycie np. bloku try – catch jest definio-

[3] Task – jednostka programowa wykonywana współbieżnie/równolegle do reszty programu w ADA. Odpowiednik wątku w innych językach programowania.

wane dopiero na poziomie realizacji w procesie wytwórczym. Projektant/programista naturalnie używa tej konstrukcji, chcąc zamknąć pewien fragment kodu w blok chroniony. Jednak blok ten niekoniecznie musi być logicznie spójny bądź atomowy. Najczęściej jest fragmentem bardziej złożonej logiki lub jest ściśle zależny od innego fragmentu kodu [1].

Jak widać w przytoczonym przykładzie, instrukcje czy też bloki instrukcji są zbyt małe, aby na nich operować. Z kolei klasy i metody okazują się być zbyt duże, gdyż operowanie na nich spowoduje myślenie kategoriami używanych mechanizmów we wcześniejszych etapach procesu wytwórczego. Brakuje więc pewnej konstrukcji (jednostki programowej), która – będąc atomową i logicznie spójną – umożliwiałaby poddanie się procesowi optymalizacji wykonania (rys. 3).



Rys. 3. Fragment struktury kodu języka obiektowego
Fig. 3. Part of a code structure of an object-oriented language

Najlepszym sposobem definicji wspomnianej konstrukcji jest użycie deklaratywów. Na ich podstawie środowisko uruchomieniowe samo rozbijałoby je na elementy wykonawcze jednostek obliczeniowych (wątki) [8]. Podział ten odbywałby się po zbudowaniu drzewa wyrażeń (tokenów, leksemów) i jego analizie. Mogłoby się okazać, że jedna jednostka programowa byłaby na tyle mało złożona obliczeniowo, iż bilans kosztów byłby ujemny (o czym szerzej w dalszej części). Środowisko wykonawcze powinno umieć również grupować tego typu jednostki programowe w jedną większą i przekazywać je do wykonania do innej jednostki obliczeniowej (inny rdzeń, inny węzeł). Mechanizm grupowania może mieć charakter: statyczny (np. w czasie kompilacji), dynamiczny (w czasie wykonywania kodu) lub hybrydowy [2]. Na dalej przedstawionym listingu pokazano fragmentu kodu, obrazującego integrację konstrukcji deklaratywnych i imperatywnych języka:

```
...
foreach (Student where Wydzial = 'WCY' and PrzysługujeStyp()) as p do {
    decimal wspolczynnik = 1.0;
    if (p.SredniDochodNaOsobe < 400) then {
```

```
            wspolczynnik *= 1.2;
        }
        else if (p.SredniDochodNaOsobe > 1000) {
            wspolczynnik *= 0.9;
        }
        logical_block {
            decimal srednia = p.WyliczSredniaStudiow();
            decimal baza = (Wydzial where Kod = 'WCY').PozycjaSredniej(srednia);
            p.PrzydzielStypendium(wspolczynnik * baza);
        }
    }
...
```

Zalety deklaratywów wprowadzonych do języków imperatywnych można dostrzec w już istniejących rozwiązaniach. Microsoft wprowadził LINQ (ang. *Language Integrated Query*) w swoim .NET Framework. Jest to mechanizm dający możliwość obsługi wielu typów danych przy tej samej konstrukcji zapytań (zbliżonej składniowo do języka SQL). Jest wygodny, szybki w użyciu i elastyczny. Nie ma znaczenia, czy operuje się na kolekcjach obiektów, XML lub na bazie danych – deklaratyw jest wciąż ten sam. Oczywiście nie ma nic za darmo. Koszt obsługi tych deklaratywów jest spory, jednak możliwości, jakie to rozwiązanie daje, rekompensują go z nadwyżką, tym bardziej że Microsoft dalej rozwija ten mechanizm m.in. o możliwość współbieżnego przetwarzania zapytań (Parallel LINQ).

3.2. Harmonogramowanie

Rozbijanie bloków kodu na oddzielne elementy wykonawcze wymaga określenia kolejności ich wykonania. Może się przecież okazać, że owszem da się wykonać dane bloki na wielu jednostkach wykonawczych (CPU, GPU), jednak niektóre z tych bloków są w pewnej mierze zależne od wyników przetwarzania innych. Efekt jest taki, że nie wszystkie bloki mogą być wykonane równocześnie. W normalnym przypadku tego rodzajami zależności musi zająć się projektant/programista, w tym rolę tę pełniłoby środowisko wykonawcze. Analiza jawnych zależności odbywałaby się z użyciem definicji deklaratywów (jeśli taka konstrukcja byłaby dopuszczona) oraz drzewa wyrażeń.

Harmonogramowanie jest więc istotne z punktu widzenia optymalizacji przetwarzania, rozumianej jako zrównoleglenie wykonywanych operacji, i odpowiada za uzyskanie poprawnego wyniku końcowego. Musi jednak uwzględniać również ukryte zależności pomiędzy jednostkami programowymi. Dana jednostka może zawierać zestaw instrukcji wraz z wywołaniem metod, a te z kolei mogą modyfikować dane. Wykonanie zrównoleglonych jednostek w wielu wątkach bez uwzględnienia tej ukrytej zależności mogłoby spowodować uzyskanie błędnego wyniku obliczeń. Idealnym rozwiązaniem byłoby zamykanie w takie jednostki programowe bloków instrukcji, które nie dają skutków ubocznych[4] (w kontekście elementów

[4] Skutek uboczny, efekt uboczny – dowolny efekt wyrażenia, który zmienia stan systemu. Z nazwy jest nieco kontrowersyjny, gdyż często zmiana ta jest właśnie zamierzona.

współdzielonych z innymi blokami, które mogłyby być realizowane równolegle), jednak często nie jest to możliwe.

3.3. Podejście procesowe

Istotnym elementem procesu wytwórczego jest przełożenie wymagań biznesowych na model projektowanego systemu. Ze względu na myślenie innymi kategoriami projektantów i biznesmenów często zdarza się, że powstały rezultat pracy zespołu odbiega od oczekiwanego. Modelowanie procesowe czy też podejście procesowe sprawia, że oba te światy zaczynają rozmawiać w tym samym języku. W świecie biznesu liczy się to, jak on funkcjonuje, a więc np. klient przychodzi do okienka, składa zamówienie, następnie jest ono wysyłane do magazynu i tam realizowane, itd. Nie myśli tu w kategoriach obiektów i zależności pomiędzy nimi. Nawet diagramy składające się z encji biznesowych (graficznej reprezentacji pojęć) są czasami zbyt dużą abstrakcją [9].

Bardzo często proces biznesowy definiuje punkty zrównoleglenia i synchronizacji, które na etapie realizacji są sprowadzane do przetwarzania sekwencyjnego. Utrzymanie przełożenia procesu na kod wymuszałoby myślenie, już na tym etapie, w kategoriach współbieżności/równoległości przetwarzania. Jest jednak na to za wcześnie. Podejście procesowe można by jednak wykorzystać. Gdyby móc przenieść bez dużych strat definicję procesu na implementację, choćby w postaci metainformacji lub nowych konstrukcji programistycznych, zyskano by nowe możliwości. Zarówno deklaratywy, jak i proces biznesowy mówią o celu, tyle że na innym poziomie abstrakcji. Ze względu na to, iż sposób rozumowania jest podobny, o wiele łatwiej byłoby przenieść definicję procesu na deklaratywy na etapie implementacji. Dzieliłyby one kod na pewne bloki danych, z których korzystać by mogły algorytmy wyszukiwania gruboziarnistej równoległości [3], oczywiście działające w tle i bez ingerencji projektanta/programisty.

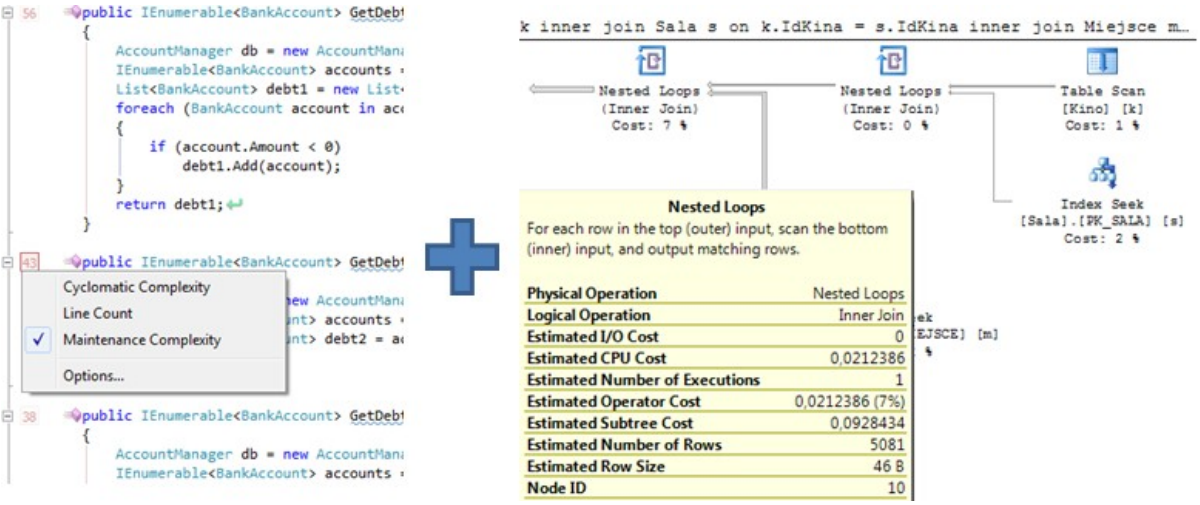
3.4. Szacowanie kosztów przetwarzania

Nie bez znaczenia dla kompletności koncepcji jest element szacowania złożoności obliczeniowej dla nowo wprowadzonych jednostek programowych. Aby rozwiązanie miało sens, oczekiwany bilans kosztów (bilans zysków i strat) musi być dodatni. Jako zysk rozumiemy tu: usprawnienie procesu wytwórczego, brak jawnie definiowanych wątków i ich synchronizacji, skupienie się na celu, a nie na sposobie jego realizacji. Do strat zaliczyć można zaś koszt zarządzania rozwiązaniem oraz koszt jego uruchomienia.

Jak wspomniano już wcześniej, środowisko powinno mieć możliwość agregowania jednostek programowych w większe grupy. Jednak operacja ta nie mogłaby być możliwa bez wcześniejszego oszacowania złożoności pojedynczej jednostki. Wracając do przykładu z listą

kont bankowych, posiadającą 10 mln elementów, i podziału przetwarzania na liczbę dostępnych rdzeni procesora, ten 1 mln elementów na rdzeń to właśnie wynik szacowania. Wynik, który mówi, iż gdy liczba elementów przekroczy wspomniane 2 mln, bilans kosztów jest dodatni lub też inaczej – większy jest zysk ze zrównoleglenia niż strata związana z jego zarządzaniem i obsługą.

Widać więc, że szacowanie musi być przeprowadzane wieloaspektowo. Z jednej strony jest szacowanie złożoności pewnych pojedynczych konstrukcji, jak np. pętle, z drugiej strony – zestaw instrukcji i bloków instrukcji oznaczonych jako pewna samodzielna jednostka programowa. Informacje potrzebne do tego typu oszacowań pobierane byłyby z samego kodu (drzewo wyrażeń) i/lub z metadanych. Nowoczesne języki programowania z reguły są również językami samoopisującymi się. Dostęp do metadanych zapewniany jest przez mechanizmy refleksji (ang. *Reflection*) i technik RTTI (ang. *Run Time Type Information*). Wykorzystanie wszystkich tych mechanizmów wraz z odpowiednimi algorytmami szacowania złożoności usprawniłoby przekładanie elementów języka na efektywne wykorzystanie zasobów [1]. Wymaganiem jest jednak wprowadzenie dodatkowych metainformacji z opisem semantycznym, które ułatwią optymalizację.



Rys. 4. Przykład analizy złożoności poleceń imperatywnych i deklaratywnych
Fig. 4. Example of complexity analysis of imperative and declarative instructions

Istniejące mechanizmy szacowania w systemach baz danych opierają się nie tylko na schemacie bazy danych, ale również na prowadzonych statystykach oraz informacjach technicznych o umiejscowieniu plików danych itp. (rys. 4). Przy wykonywaniu polecenia SQL może być generowanych wiele planów wykonania, następnie każdy z nich jest sprawdzany pod kątem złożoności i czasochłonności wykonania, a później wybierany jest plan optymalny przy ustalonej pewnej funkcji celu [5]. Taki sam mechanizm można wykorzystać w omawianej koncepcji. Oznacza to, że z założenia nie ma jednego algorytmu podziału jednostek programowych. Generowanych jest wiele „pomysłów" środowiska wykonawczego na rozproszenie przetwarzania, wszystkie są szacowane i wybierany jest najlepszy.

4. Podsumowanie

Niniejszy artykuł jest zaprezentowaniem koncepcji uproszczenia procesu wytwórczego przez zastosowanie w językach imperatywnych konstrukcji deklaratywnych do rozpraszania i optymalizacji przetwarzania danych. Jak przedstawiono we wcześniejszych rozdziałach, moc deklaratywów jest nie do przecenienia. Istnieją już takie języki, które maksymalnie wykorzystują dostępne zasoby i mają się przy tym bardzo dobrze (np. SQL) [6, 10]. Poza językami deklaratywnymi nie są póki co dostępne konstrukcje, które umożliwiałyby dynamiczne zrównoleglenie przetwarzania na etapie wykonawczym. Obecnie istniejące środowiska programistyczne wymagają od projektantów/programistów myślenia w kategoriach współbieżności już na etapach modelowania i projektowania, ponadto wymagają jawnej implementacji takiej współbieżności. Jest to rozwiązanie czasochłonne, wymagające wiedzy i doświadczenia, a przy tym niezbyt bezpieczne.

Pierwszym krokiem realizacji przedstawionej koncepcji będzie budowa konstrukcji językowych, które – poza uwolnieniem projektanta/programisty z jawnego podziału zadań na wątki, ich implementowania i synchronizowania – sprawiłyby, że zacząłby on myśleć bardziej w kategoriach biznesowych niż technicznych. Istnieje obecnie wiele generatorów aplikacji biznesowych, w których główny nacisk kładzie się na to, co chcemy osiągnąć, i na opis przy użyciu deklaratywów. Jednak są to rozwiązania jednoprocesowe i jednowątkowe, a użycie deklaratywów sprowadza się do oznaczania reguł walidacji i opisu semantycznego. Kolejnym krokiem, który należałoby wykonać w tych mechanizmach, jest rozszerzenie istniejącego opisu o informacje ułatwiające optymalizację przetwarzania. Istotnym elementem byłoby tu wprowadzenie nowej jednostki programowej, która w prosty sposób poddałaby się analizie, szacowaniu i zrównolegleniu na etapie wykonawczym.

Ostatnim krokiem budowy rozwiązania będzie budowa mechanizmów szacowania i harmonogramowania. Są one istotne z punktu widzenia opłacalności przeprowadzania pewnych operacji oraz uzyskania poprawnego wyniku końcowego. Jak wspomniano w artykule, z założenia nie istniałby jeden algorytm podziału jednostek programowych. Deklaratywna konstrukcja powinna umożliwiać generację wielu planów wykonania i wyboru najlepszego.

BIBLIOGRAFIA

1. Chudy M.: Wybrane zagadnienia podstaw informatyki. Wojskowa Akademia Techniczna, Warszawa 2005.
2. Hopcroft J., Ullman J.: Wprowadzenie do teorii automatów, języków i obliczeń. PWN, Warszawa 2003.

3. Siedlecki K.: Algorytmy wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi. Portal Zachodniopomorskiej Biblioteki Cyfrowej, http://zbc.ksiaznica.szczecin.pl/dlibra/doccontent?id=4621&dirids=32, 2008.
4. Stencel K.: Półmocna kontrola typów w językach programowania baz danych. PJWSTK, Warszawa 2006.
5. Subieta K.: Teoria i konstrukcja obiektowych języków zapytań. PJWSTK, Warszawa 2004.
6. Diaconescu R.: Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness. Department of Computer and Information Science, Norwegian University of Science and Technology, http://www.idi.ntnu.no/grupper/su/publ/phd/roxana-thesis.pdf, 2002.
7. Hellerstein J. M.: The Declarative Imperative, Experiences and Conjectures in Distributed Logic. University of California, Berkeley, http://db.cs.berkeley.edu/papers/sigrec10-declimperative.pdf, 2010.
8. Koszela J. et al.: Executive environment of Distributed Object Database MUTDOD. Studia Informatica, Vol. 32, No. 3B(99), Wydawnictwo Politechniki Śląskiej, Gliwice 2011, s. 77÷87.
9. Fahland D. et al.: Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. 10th Workshop on Business Process Modeling, Development, and Support (BPMDS 2009) and 14th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2009), Amsterdam 2009, http://www.mendling.com/publications/09-EMMSAD-Declare.pdf.
10. http://www.odbms.org/odmg/.

Wpłynęło do Redakcji 15 stycznia 2012 r.

Abstract

Current hardware improvements are not followed by improvements of methods, technics and development tools. During design and modelling process designer needs to take into consideration concurrent/parallel language constructions (Fig. 1) to develop efficient solutions. It's too early to think in such categories. By extending imperative properties of programming languages with declarative constructions, it would be eliminated. Idea is to maximize the use of power of a changing hardware and/or environment without the need of recompiling code. Extensions mentioned above (in the form of declaratives or program units) would allow execution engine to distribute processing among available processing nodes or whole distributed

system (Fig. 2, 3). Effectiveness of whole computation process (optimization) is taken into consideration during that distribution. The power of the declarative constructions, and the reason that distribution could be possible, is that they are logically closed. By introducing new constructions to code, flexibility and expressiveness of a language is much improved. The first step and the example to achieve that was introducing lambda expressions.

Declarative constructions and program units should be used in natural way as other programming constructions (i.e. try – catch). By connecting development with process approach, building even complex systems would be easier. During gathering the requirements and defining business process, points of split and merge in processing are very often defined. Moving such information directly to code, under the form of declaratives, would tell execution engine how to effectively manage program units. The main area of interest is business applications. Because of economic reasons it is more accurate to build application that can adapt to changes in an environment than modifying the code (risk of regression, time consuming, higher cost).

Declaratives during processing can produce several execution plans before they are executed. Execution engine computes complexity and analyse all of them, then choose the optimal one. But this is only one part of a solution. Another one, and also very important, is the scheduling unit. Multi-aspect estimation and balance of costs and benefits are the core functionality (Fig. 4), but very often program units are connected with each other through dependencies. Some program units can depend on another one. Ability to find such dependency is crucial to build effective solution.

Adresy

Marcin DĄBKIEWICZ: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. Kaliskiego 2, 00-908 Warszawa 49, Polska, mdabkiewicz@wat.edu.pl.
Jarosław KOSZELA: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. Kaliskiego 2, 00-908 Warszawa 49, Polska, jkoszela@wat.edu.pl.