JAROSŁAW FRANCIK

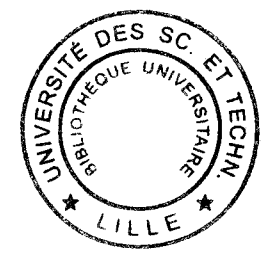
22 MARCA 1999

ŚLEDZENIE PRZEPŁYWU DANYCH DLA POTRZEB ANIMACJI ALGORYTMÓW

ROZPRAWA DOKTORSKA

PROMOTORZY:

PROF. DR INŻ. STEFAN WĘGRZYN
PROF. JEAN-MARC TOULOTTE



INSTYTUT INFORMATYKI POLITECHNIKI ŚLĄSKIEJ, GLIWICE, POLSKA
LABORATOIRE D'AUTOMATIQUE I^{3}D, UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE, FRANCE
1999

Mojej Żonie, Kasi

SPIS TREŚCI

PODZIĘKOWANIA

Długo przygotowywałem się do tego miłego obowiązku, jakim jest złożenie podziękowań tym wszystkim, których wysiłek przyczynił się do tego, że mój ponad pięcioletni okres pracy na Politechnice Śląskiej oraz na Uniwersytecie w Lille mogę ukoronować w formie niniejszej pracy.

Przede wszystkim pragnę podziękować promotorom pracy, profesorowi Stefanowi Węgrzynowi oraz profesorowi Jean-Marc Toulotte'owi, nie tylko za ich wkład merytoryczny, ale także ogromną pomoc okazaną na płaszczyźnie organizacyjnej. Dziękuję także dyrektorom jednostek, w których pracę realizowałem: profesorom Andrzejowi Grzywakowi i Christianowi Vasseurowi, a także profesorowi Stanisławowi Kozielskiemu. Bez ich zaangażowania w kwestie organizacyjne nie byłbym w stanie zrealizować swoich planów.

Zwykłe słowo „dziękuję" jest zupełnie niewystarczające w stosunku do doktora Przemysława Szmala: mojego szefa, współpracownika i przyjaciela, który niestrudzenie czytał wczesne (i nie tylko) wersje tekstu i przyczynił się w ogromnym stopniu do jego udoskonalenia. Przez cały czas był też niestrudzony w skutecznym dopingowaniu mnie do pracy, do której – jako mój szef – dawał mi doskonałe warunki organizacyjne. Tekst pracy, a właściwie jego obcojęzyczne wersje, czytał też wiele razy pan Rachid Ikni z Uniwersytetu w Calais – również jego uwagi, tak merytoryczne, jak i techniczne a nawet edycyjne, spowodowały, że tekst pracy jest teraz dużo lepszy (i bardziej przypomina pracę naukową).

Chciałbym także podziękować zespołowi moich najbliższych współpracowników, do których należą, bądź należeli: Krzyś Dobosz, Mateusz Nowak, Paweł Gonera, Artur Migas, Ania Tomaszewska, Krzychu Wójcik oraz autorzy wczesnych wersji *SANALa*, a także Frédéric Van de Veire. Wszyscy mają swój wkład w tę pracę. Dziękuję też koleżankom i kolegom z Instytutu Informatyki Politechniki Śląskiej oraz z Laboratorium Automatyki I^3D Uniwersytetu w Lille, a w szczególności Olivierovi Losson, Jean-Marcowi VanNobelowi i Erykowi

Czesnałowiczowi za pomoc i przyjaźń, z jakimi spotkałem się z ich strony w trakcie pobytu we Francji, oraz profesorowi T. Czachórskiemu, który nie tylko tłumaczył dla mnie teksty na język francuski, ale także chętnie dzielił się ze mną swoją wiedzą o Francji i doświadczeniami z życia w tym pięknym kraju. Nie mogę także pominąć pani K. Wrześniowskiej, która wprowadziła mnie w świat języka francuskiego, ani mojej teściowej, od której tak wiele się nauczyłem o kraju jej dzieciństwa.

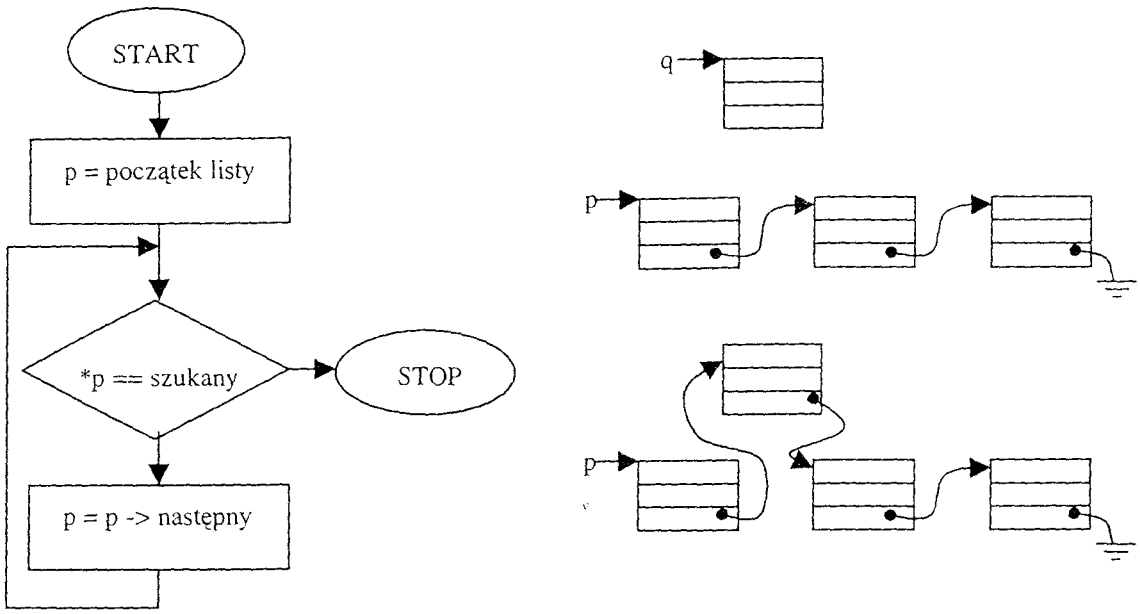
Trudno byłoby wymienić wszystkich, którzy dzięki twórczym dyskusjom na różne tematy przyczynili się do ulepszenia tej pracy. Wymienię tylko kilka osób: profesora Adama Mrózka, doktorów Marka Konopkę i Lecha Znamirowskiego, a także Wojciecha Mikanika i Michała Kolano. Dziękuję także wszystkim osobom, które umożliwiły mi pracę od strony technicznej, dbając o dobrą kondycję sprzętu komputerowego i załatwiając miliony spraw biurowych, a szczególnie Krzysiowi Podstawie, który nie tylko był nieocenionym pracownikiem technicznym, ale też moim dobrym przyjacielem.

Na końcu, ale jakże ważne podziękowania złożyć pragnę mojej rodzinie, która znosić musiała moje nastroje w trakcie realizacji prac. Dotyczy to zarówno rodziców (faza wcześniejsza), jak i mojej żony (faza późniejsza). Moja żona, Kasia, ma też swój bezpośredni wkład w pracę: wykonała część ilustracji, których szkice wielokrotnie zmieniałem.

1
WSTĘP

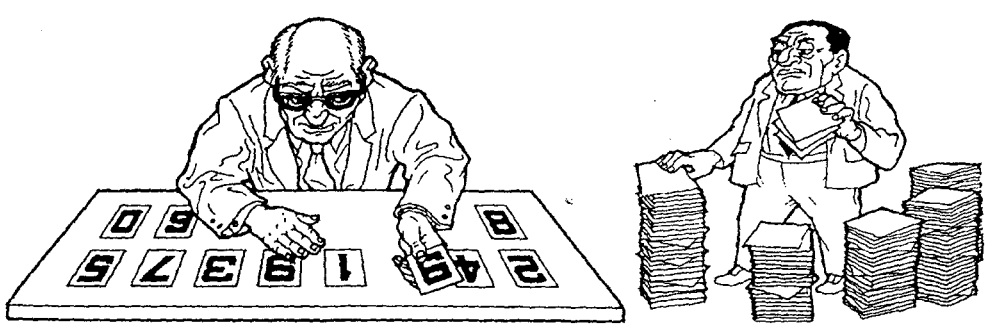
Zrozumienie sposobu działania algorytmów jest jednym z najważniejszych wymagań stawianych twórcom oprogramowania. Jest ono nieodzownym elementem zarówno w dydaktyce informatyki, jak i w praktyce inżynierskiej.

Sposób opisu algorytmu ma bardzo duży wpływ na stopień zrozumienia jego działania. Opis algorytmu w języku formalnym jest sposobem najbardziej precyzyjnym, jednak równocześnie najgorzej dostosowanym do ludzkiego systemu percepcji [14, 15]. Nie ułatwia on zrozumienia algorytmu przez człowieka. Opis algorytmu w języku naturalnym jest znacznie łatwiejszy do zrozumienia (choć być może nie tak precyzyjny), jednak w dalszym ciągu zupełnie niewystarczający, gdy w grę wchodzą bardziej skomplikowane algorytmy. Oba sposoby opisu to sposoby tekstowe; przedstawiają one opisywany przedmiot w sposób analityczny, krok po kroku, nie ułatwiają natomiast oglądu syntetycznego, niezbędnego dla zrozumienia całości problemu. Znacznie lepszym sposobem jest opis za pośrednictwem środków wizualnych; jest on znacznie bliższy modelowi percepcji człowieka (ewolucyjnie starszy od opisu językowego) i umożliwia syntetyczne spojrzenie na całość problemu, nie wykluczając wglądu w szczegóły [16, 17]. Obserwacja pracy zarówno nauczycieli informatyki, jak i inżynierów, przynosi spostrzeżenie, że bardzo często posługują się oni rysunkami po to, by ułatwić zrozumienie algorytmu. Przykładami często stosowanych graficznych abstrakcji algorytmów są schematy blokowe (tzw. sieci działań) oraz rysunki przedstawiające struktury danych (rys. 1.1). Rysunki mogą operować różnymi poziomami abstrakcji, mogą też stosować specyficzne reprezentacje graficzne, nawiązujące do obserwowalnych realiów, wyobrażeń bądź też metafor pomocnych przy opisie problemu (por. rys. 1.2). Dobór środków prezentacji zależy też od charakteru, ilości i organizacji danych opisujących problem.



Rys. 1.1. Przykłady często stosowanych graficznych abstrakcji algorytmów: schemat blokowy wyszukiwania danych na liście i rysunek ilustrujący wstawianie elementów

Kilkanaście ostatnich lat rozwoju informatyki było okresem bezprecedensowego rozwoju rozwiązań graficznych w zakresie pośrednictwa między człowiekiem i komputerem. Graficzne interfejsy użytkownika (ang. *graphical user interface, GUI*) są obecnie powszechnie stosowanym standardem, że wymienimy tylko *Microsoft Windows*, *XWindows* czy oprogramowanie komputerów *Macintosh*. Okazały się one niezmiernie skuteczne we wszystkich dziedzinach zastosowań technik komputerowych. Wizualizacja oprogramowania jest wyrazem tej tendencji w zakresie konstrukcji oprogramowania i dydaktyki informatyki. Umożliwia uwidocznienie określonych cech algorytmów za pomocą odpowiednio dobranej reprezentacji graficznej. Wizualizacja algorytmów może pozwalać nie tylko na tworzenie statycznych modeli, ale także dynamicznych **symulacji procesu wykonywania algorytmu**. Taką właśnie wizu-



Rys. 1.2. Zastosowanie metafory dla uwidocznienia różnicy między sortowaniem ciągu a sortowaniem pliku [198, str. 69-70]

alizację, skonstruowaną przy wykorzystaniu dynamicznie zmieniającego się obrazu, nazywamy **animacją algorytmów**. Jest środkiem do eksploracji działania i zachowania algorytmów. Umożliwia ona wgląd w sposób działania i zachowania się algorytmów, czy szerzej – oprogramowania, używając języka najbardziej zrozumiałego dla człowieka – języka obrazów. Pozwala dostrzec wiele aspektów działania algorytmów trudno zauważalnych w inny sposób, czy wręcz niemal całkiem nieuchwytnych. Stanowią one niezwykle użyteczny poligon doświadczalny, pozwalający śledzić dynamiczne zachowanie programów. W sposób fundamentalny zmienia to i ulepsza sposób rozumienia algorytmów i sposób myślenia o nich [11, 73], daje też rewelacyjne rezultaty w zastosowaniach dydaktycznych [50, 52, 53, 56 – 58].

W ciągu minionych lat wizualizacja algorytmów i programów przyciągała sporo uwagi [11]. Stworzono wiele systemów dostarczających środki do jej realizacji. Skuteczna wizualizacja powinna jednak wychodzić poza izomorficzne odwzorowanie struktur danych w obrazy graficzne, i odzwierciedlać semantykę algorytmu na odpowiednim poziomie abstrakcji. Najczęściej wymaga to wprowadzenia dodatkowych, zorientowanych na użytkownika informacji dotyczących tych aspektów oprogramowania, które projektant wizualizacji uznaje za istotne w danym kontekście. Reprezentują one wiedzę projektanta na temat poddawanego wizualizacji systemu. Z tego powodu uważa się, że proces przygotowania takich wizualizacji wymaga znacznego wysiłku ze strony osoby przygotowującej tę wizualizację i nie da się skutecznie zautomatyzować [9 – 12, 73].

Celem niniejszej pracy jest wykazanie, że możliwe jest ograniczenie wysiłku związanego z przygotowaniem wizualizacji algorytmów charakteryzującej się wysokim stopniem abstrakcji, a nawet częściowe zautomatyzowanie tego procesu. Zaproponowana zostanie oryginalna metoda animacji algorytmów, korzystająca z techniki śledzenia przepływu danych. Przedstawione też zostanie jej zastosowanie w systemie animacji algorytmów *Daphnis*. System ten może być wykorzystany zarówno w celach dydaktycznych, jak i inżynierskich. Mając nadzieję, że rola systemu *Daphnis* nie ograniczy się do pozostania poligonem doświadczalnym dla dalszych prac badawczych nad wizualizacją algorytmów, udostępniamy go szerokim kręgom potencjalnych użytkowników za pośrednictwem sieci Internet. Pełna wersja systemu jest dostępna pod adresem:

http://www-zo.iinf.polsl.gliwice.pl/~jfrancik/aa/download.html

1.1 UKŁAD TREŚCI

Treść pracy jest podzielona na dziewięć rozdziałów.

Pierwsze trzy rozdziały stanowią wprowadzenie do zagadnienia wizualizacji oprogramowania oraz przedstawiają obecny stan badań w tej dziedzinie. I tak, w rozdziale pierwszym zdefiniujemy najistotniejsze pojęcia, którymi posługiwać się będziemy w dalszej części pracy. W punkcie 1.3 (str. 23) sformułujemy tezę pracy. Rozdział drugi został poświęcony głębszej analizie najistotniejszych z naszego punktu widzenia problemów związanych z wizualizacją oprogramowania. W rozdziale trzecim przedstawimy aktualny stan badań w dziedzinie wizualizacji oprogramowania. Dokonamy przeglądu istniejących rozwiązań, a także podejmiemy próbę ich ogólnej klasyfikacji.

W rozdziale czwartym wprowadzimy oryginalną metodę animacji algorytmów opartą na śledzeniu przepływu danych. Sformułujemy trzy zasadnicze założenia skutecznej wizualizacji, charakteryzującej się wysokim poziomem abstrakcji, i pokażemy, że można je spełnić stosując naszą metodę. Do opisu proponowanego algorytmu skorzystamy z formalizmu sieci Petriego. Przedstawimy kilka wersji algorytmu; jedna z nich, zwana algorytmem 1,5-krokowym, trójkolorowym została zaimplementowana w systemie *Daphnis*.

Kolejne cztery rozdziały będą poświęcone systemowi *Daphnis*. I tak, ogólna charakterystyka systemu jest przedmiotem rozdziału piątego. Przedstawiona jest tam też geneza systemu. W rozdziale szóstym przedstawimy interaktywne środowisko systemu. Rozdział ten zawiera analizę sylwetki potencjalnego użytkownika, wyjaśnia założenia projektowe, którymi się kierowaliśmy, a także opis funkcjonalny systemu, umożliwiający rozpoczęcie pracy z systemem *Daphnis*. Tematem rozdziału siódmego jest implementacja systemu *Daphnis*. Omówimy architekturę maszyny projekcyjnej, której koncepcja nawiązuje do modelu *POST*, warstwę pośrednictwa system – poddawany wizualizacji program, w której skorzystaliśmy ze standardu *Active-X*, oraz przedstawimy *Daphnis* jako system otwarty. W rozdziale ósmym pokażemy przykłady wykorzystania systemu. Przedstawimy pełne teksty źródłowe poddawanych wizualizacji programów, wraz z odpowiednimi skryptami konfiguracyjnymi. Przykłady ilustrują kolorowe plansze.

Wnioski i perspektywy na przyszłość stanowią treść ostatniego, dziewiątego rozdziału.

1.2 DEFINICJE

Zdefiniujemy teraz najistotniejsze pojęcia, którymi posługiwać się będziemy w dalszej części pracy. Wiele zasygnalizowanych tu problemów znajdzie rozwinięcie w rozdziale 2.

1.2.1 ALGORYTMY I ICH REALIZACJA

Algorytm to taki zbiór operacji, że po ich wykonaniu otrzymuje się rozwiązanie dowolnego zadania z określonej klasy zadań, przy spełnieniu określonych warunków wstępnych. Operacje algorytmu powinny być wykonywane w określonym porządku [194, 195, 197, 198]. Dowolną procedurę lub program korzystający z algorytmu w celu rozwiązania określonego zadania (najczęściej przetworzenia danych wejściowych na określone dane wyjściowe) nazywamy **realizacją** albo **implementacją** algorytmu. Algorytmy istnieją niezależnie od swoich implementacji.

Program komputerowy jest tylko jednym z wielu możliwych przykładów różnych realizacji algorytmów, zarazem jedyną, która pozostaje w zakresie naszych zainteresowań w ramach tej pracy. Dlatego za realizację (implementację) algorytmu będziemy odtąd uważać program komputerowy (lub jego fragment). Zauważmy, że nie jest ścisłe określenie *wykonanie algorytmu*: to nie algorytmy, ale programy je realizujące mogą być wykonywane. Mimo to stosujemy pojęcie wykonania algorytmu, gdyż jest ono użytecznym skrótem myślowym.

Określony algorytm może być realizowany na różne sposoby. Implementacja może zależeć, między innymi, od użytego języka programowania, struktury systemu operacyjnego czy od architektury komputera. Najistotniejsze różnice między różnymi implementacjami wynikają z kolejności wykonywania operacji.

Mówimy o **sekwencyjnej** realizacji algorytmu, jeżeli w każdej chwili wykonywana jest tylko jedna operacja (instrukcja), a wykonanie każdej operacji rozpoczyna się po zakończeniu wszystkich poprzednio wykonywanych operacji, i kończy przed rozpoczęciem wykonania wszystkich operacji po niej następujących (rys. 1.3a), to znaczy:

$$\forall i \in [1,n] \qquad \tau_k(o_i) \leq \tau_p(o_{i+1}) \tag{1}$$

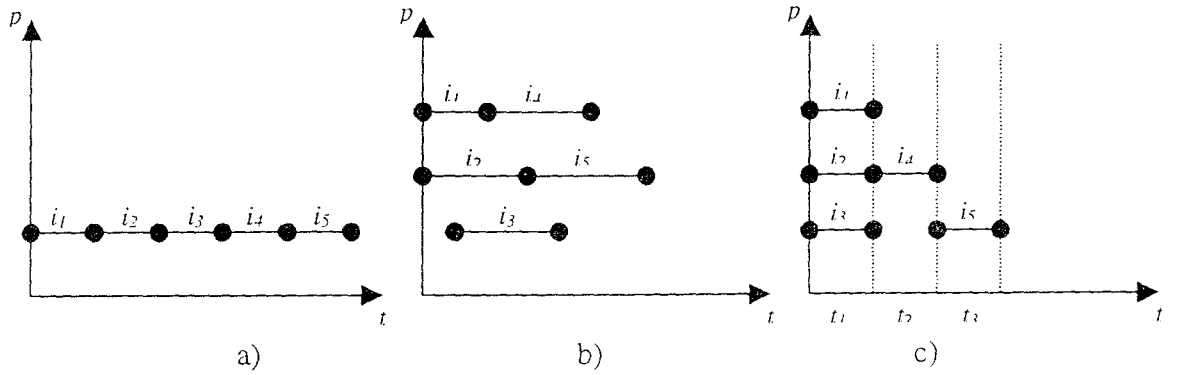
gdzie:

o_i – i-ta operacja w algorytmie,

$\tau_p(o_i)$ – czas rozpoczęcia operacji o_i,

$\tau_k(o_i)$ – czas zakończenia operacji o_i,

przy czym zakładamy, że operacje algorytmu zostały ponumerowane od *1* do *n+1* włącznie. Żadne dwie operacje nie są wykonywane w tym samym czasie. Poszczególne instrukcje są



Rys. 1.3. Wykresy Gantta obrazujące sposoby realizacji programów:
a) sekwencyjny, b) równoległy asynchroniczny, c) równoległy synchroniczny

wykonywane w ściśle określonej kolejności. Jest to najprostszy, i co za tym idzie, najpopularniejszy sposób pisania programów. Algorytmy tak zaimplementowane są na ogół określane mianem *algorytmów szeregowych*. Jednak dla niemal każdego algorytmu można wskazać operacje, które można wykonać w tym samym czasie. W tym trybie realizacji, zwanym **równoległym**, operacje mogą być wykonywane zanim operacje je poprzedzające zostaną ukończone (rys. 1.3b). Zależność (1) nie jest spełniona, to znaczy:

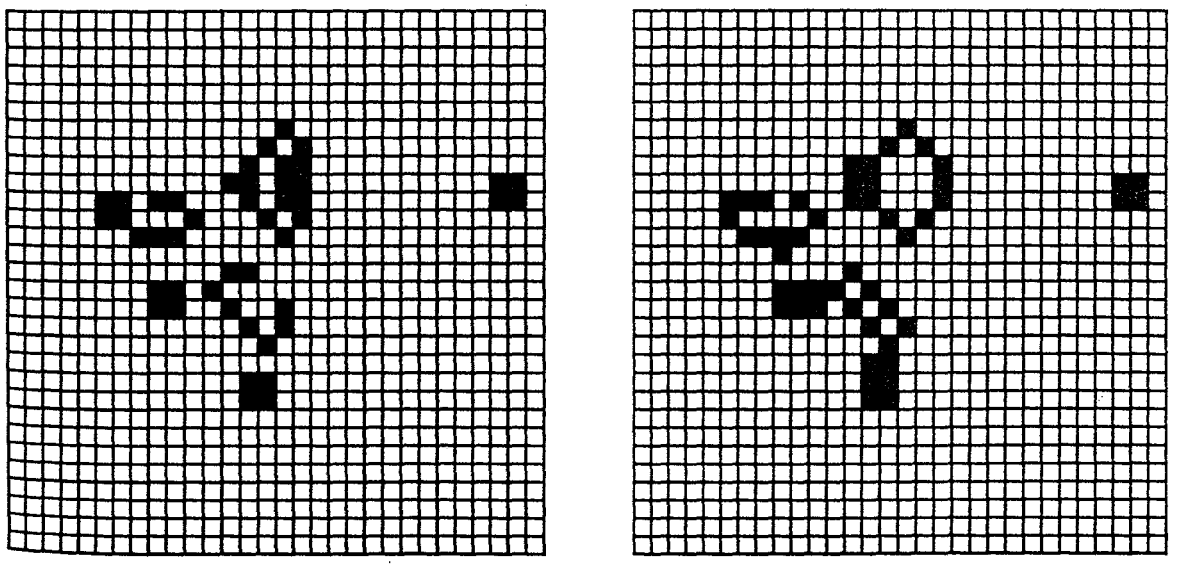
$$\exists i \in [1,n] \qquad \tau_k(o_i) > \tau_p(o_{i+1})$$

W większości równoległych architektur komputerowych operacje są wykonywane w kilku wątkach. W obrębie jednego wątku operacje są wykonywane szeregowo, natomiast operacje należące do różnych wątków mogą, chociaż nie muszą, być wykonywane w tym samym czasie (równolegle). Co więcej, sekwencja, w której wykonywane są operacje należące do różnych wątków (zwana czasem **scenariuszem** [165]), jest niezdeterminowana. Taki rodzaj równoległości jest nazywany **asynchronicznym**. Istnieją algorytmy zaprojektowane z myślą o takiej właśnie, równoległej i asynchronicznej metodzie realizacji; nazywamy je *algorytmami równoległymi (asynchronicznymi)*. Jednak każdy z nich może być również wykonany w trybie szeregowym. Tak więc określenia *szeregowy* i *równoległy* w odniesieniu do algorytmu nie są ścisłe: każdy z nich może być wykonany zarówno szeregowo, jak i równolegle, a tylko jedne z nich są lepiej dostosowane (zoptymalizowane) do wykonania szeregowego, inne zaś – do równoległego. Dobrym przykładem jest algorytm szybkiego sortowania (ang. *quicksort*), z dobrym skutkiem zaimplementowany sekwencyjnie przez Hoare'a [166]. Wprowadzenie niewielkich modyfikacji umożliwiło utworzenie takich wersji tego algorytmu, które w trybie realizacji równoległej są bardzo wydajne [167, 168].

Tryb realizacji **równoległy synchroniczny**, w przeciwieństwie do asynchronicznego, jest niemal nieznany w inżynierii programowania. W tym typie równoległości sekwencja równolegle wykonywanych operacji jest ściśle zdeterminowana (rys. 1.3c). Niektóre operacje muszą być wykonywane w tym samym czasie. Wiele algorytmów przeznaczonych do synchronicznego wykonania nie może być zrealizowanych asynchronicznie ani sekwencyjnie bez istotnych modyfikacji. Klasycznym przykładem rozwiązania synchronicznego jest **automat komórkowy** [169 – 171]. W ogólnym przypadku stanowi on macierz identycznie zaprogramowanych elementarnych automatów – komórek. Najczęściej przyjmuje się dwuwymiarową macierz *A[N×M]*, chociaż rozpatruje się także i inne konfiguracje. Stan każdej komórki $a_{i,j}$ w kroku $n+1$ jest funkcją stanów komórek w kroku poprzednim n:

$$\forall(0 < i \le N, 0 < j \le M): \quad (a_{i,j})_{n+1} = f((a_{1,1})_n, (a_{1,2})_n, ..., (a_{N,M})_n)$$

Najczęściej dla obliczenia nowego stanu komórki bierze się pod uwagę stany komórek znajdujących się w tzw. sąsiedztwie – które to pojęcie mniej lub bardziej odpowiada fizycznej bliskości komórek. Najlepiej znanym przykładem automatu komórkowego jest tzw. **gra w życie** (rys. 1.4), której zasady opracował pod koniec lat sześćdziesiątych John Horton Conway z Uniwesrytetu w Cambridge w Anglii [172 – 175]. Komórki w kolejnych krokach ewoluują w trudny do przewidzenia sposób, który przypomina proces rozwojowy kolonii organizmów żywych. Wartości stanów komórek należą do zbioru *{0, 1}*, przy czym komórki w stanie



Rys. 1.4. Dwa kolejne stadia gry w życie. Komórka przeżywa, gdy w jej sąsiedztwie znajduje się 2 lub 3 żywe komórki. Nowa komórka powstaje, gdy w jej sąsiedztwie znajdują się dokładnie 3 żywe komórki.

o wartości *1* określa się jako *żywe*, a te w stanie o wartości *0* – jako martwe. Stan komórki w kolejnym kroku określa się w zależności od jej poprzedniego stanu oraz stanu ośmiu komórek sąsiadujących, według następujących reguł:

1. Jeśli komórka jest martwa, to staje się żywa, jeśli w jej sąsiedztwie znajdują się dokładnie trzy żywe komórki.
2. Jeśli komórka jest żywa, to pozostaje w tym stanie tylko pod warunkiem, że w jej sąsiedztwie znajdują się dwie lub trzy żywe komórki.

Warunkiem prawidłowego określenia nowego stanu automatu jest *jednoczesne* obliczenie wartości stanu wszystkich komórek. Wszystkie zmiany muszą nastąpić w tym samym czasie – w przeciwnym przypadku konfiguracja automatu w kolejnym kroku zostałaby zakłócona. Taka jest naturalna, synchroniczna realizacja algorytmu. Realizacja asynchroniczna lub sekwencyjna wymaga zastosowania dodatkowej struktury danych, tak by macierz reprezentująca krok $n+1$ była zapamiętywana w innym miejscu, niż macierz reprezentująca krok n.

1.2.2 WIZUALIZACJA OPROGRAMOWANIA

Terminy takie, jak wizualizacja algorytmów, programów bądź oprogramowania, a także animacja algorytmów, były wielokrotnie definiowane i wykorzystywane w literaturze.

Wizualizacja oprogramowania (ang. *software visualisation*) jest często definiowana jako przedstawienie wybranych cech oprogramowania przy zastosowaniu odpowiedniej reprezentacji graficznej [12, 105, 107]. Słowo „wizualizacja" jest często zawężane do obrazów widzialnych. Słowo to jest stosunkowo świeżym zapożyczeniem w języku polskim, i próżno by go szukać w większości słowników, a nowsze spośród nich podają definicję czysto techniczną [209 – 211]. Dlatego sięgniemy do słownika Webstera, który wizualizację definiuje jako „formowanie mentalnego obrazu (oryg. ang. *mental image*) czegoś aktualnie niewidocznego, abstrakcji" [212]. Tak więc wizualizacja nie jest ograniczona do reprezentacji graficznej, gdyż taki obraz mentalny może być formowany jako rezultat danych wejściowych uzyskanych z dowolnego ze zmysłów. Na przykład, **udźwiękowienie** (ang. *auralisation*), jest rodzajem wizualizacji, która jako medium komunikacyjne wykorzystuje dźwięk zamiast obrazu [19, 20, 134].

W zasadzie wszystkie formy zapisu programów stosowane w ciągu ostatnich 50 lat były wizualne: dotyczy to nie tylko tekstu źródłowego programu i procesu czytania go. Nawet programiści używający prostego, monochromatycznego terminala tekstowego typu VT-100, z czcionką

stałej szerokości, otrzymywali swój ułatwiający zrozumienie programów obraz mentalny na podstawie wcięć w tekście źródłowym [24, 25] oraz proporcji rozmiarów bloków tekstu [64].

Ostatecznie, **wizualizację oprogramowania** będziemy definiować jako zastosowanie różnorodnych, zaawansowanych technik komputerowych do znaczącego zwiększenia poziomu zrozumienia programów komputerowych przez człowieka [5]. Techniki wykorzystywane w ramach wizualizacji obejmują, między innymi, grafikę komputerową, typografię, fotografię, animację, odtwarzanie i generowanie dźwięku, muzyki i mowy, a także kinematografię (odtwarzanie obrazu video). Techniki te są łącznie znane jako multimedia.

Wizualizacja może być **dynamiczna** lub **statyczna**, w zależności od tego, czy prezentowane obrazy zmieniają się w trakcie projekcji, czy też nie. Przykładami wizualizacji statycznej są: ładne drukowanie tekstów źródłowych programów (ang. *pretty-printing*), diagramy przepływu sterowania i danych lub schematyczne rysunki struktur danych. W niniejszej pracy interesować nas będzie jednak wyłącznie wizualizacja dynamiczna.

Termin „wizualizacja oprogramowania" wprowadzili po raz pierwszy Price, Baecker i Small [5], zastępując szeroko stosowane w literaturze pojęcie „wizualizacji programów" (ang. *program visualisation*). W ich ujęciu (które też przyjmujemy), **wizualizacja programów** jest odmianą wizualizacji oprogramowania koncentrującą się na tych aspektach wizualizowanego oprogramowania, które łączą się z konkretną realizacją, to znaczy – z programem komputerowym. Przedmiotem wizualizacji jest przede wszystkim rzeczywiście zastosowany kod programu. W przeciwieństwie do tego, **wizualizacja algorytmów** uwidacznia raczej cechy samego algorytmu, w oderwaniu od jego konkretnej realizacji. Jest to wizualizacja opisu oprogramowania dokonanego na wysokim, abstrakcyjnym poziomie. Obydwie kategorie są komplementarne, i istnieje kontinuum możliwych wizualizacji pośrednich, zależnie od stopnia, w jakim abstrahują one od konkretnej realizacji algorytmu. Zauważmy też, że mając skuteczne środki do wizualizacji algorytmów mamy tym samym środki do wizualizacji programów: ta ostatnia wymaga bardzo podobnych narzędzi, łatwiej jednak jest wizualizować programy niż algorytmy.

1.2.3 ABSTRAKCJA W WIZUALIZACJI OPROGRAMOWANIA

Pojęcie **abstrakcji** jest w Słowniku Języka Polskiego [209] zdefiniowane jako „działanie myślowe polegające na wyodrębnieniu cech istotnych, stałych jakiegoś przedmiotu lub zjawiska i rozpatrywaniu ich w oderwaniu od cech nieistotnych, przygodnych". Abstrakcja w wi-

zualizacji oprogramowania polega na ignorowaniu wybranych aspektów poddawanego wizualizacji programu lub algorytmu, jeśli nie stanowią one efektywnego narzędzia ułatwiającego zrozumienie oprogramowania. Z drugiej strony, obejmuje też wprowadzenie pewnych elementów, które nie są obecne ani w programie, ani w jego strukturach danych, ale są istotne dla zrozumienia oprogramowania, nawet jeśli nie niosą ze sobą wprost żadnych informacji o sposobie realizacji algorytmu przez komputer.

Prostym miernikiem **poziomu abstrakcji** w wizualizacji oprogramowania jest to, w jakim stopniu prezentacja jest izomorficzna w stosunku do komponentów oprogramowania, które reprezentuje [7, 9 – 12], na przykład, czy przedstawiane struktury danych można odtworzyć na podstawie ich reprezentacji graficznej tak samo łatwo, jak reprezentacje te utworzono na podstawie zawartości struktur danych. Wizualizacje algorytmów, w przeciwieństwie do wizualizacji programów, często znacznie wychodzą poza izomorficzne odwzorowanie programu lub danych na reprezentację graficzną. Charakteryzują się one stosunkowo wysokim poziomem abstrakcji; były nawet w ten sposób definiowane [1, 5].

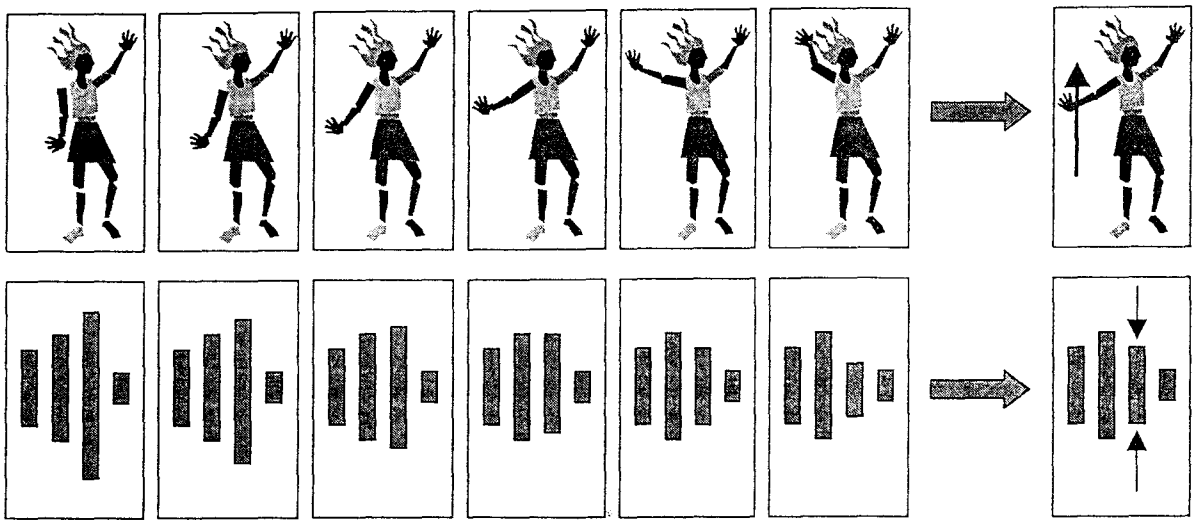
Wysoce abstrakcyjne wizualizacje dostarczają dodatkowych informacji semantycznych, niedostępnych wprost w tekście programu. Chodzi o zorientowane na użytkownika informacje dotyczące aspektów uznanych przez projektanta wizualizacji za istotne w kontekście opisu konkretnego oprogramowania, reprezentujące wiedzę projektanta na temat tego oprogramowania. Informacje te nazywamy **zawartością intencyjną** (ang. *intention content*) [12].

1.2.4 ANIMACJA ALGORYTMÓW

Termin **animacja** był – niestety – często nadużywany. W konsekwencji wiele systemów, które reklamowano jako udostępniające animację, ograniczało się do podświetlania aktualnie wykonywanej linii programu lub prostych, skokowych zmian sposobu wyświetlania elementów graficznych [12].

Obecnie animacja należy do standardu przemysłowego w zastosowaniach multimedialnych, i polega na szybkim wyświetlaniu serii stopniowo zmieniających się w czasie obrazów, zwanych kadrami animacji. Zmiany pomiędzy następującymi po sobie kadrami są na tyle małe, a tempo wyświetlania kadrów na tyle duże, że uzyskuje się iluzję ciągłego, płynnego ruchu – jak w konwencjonalnym filmie animowanym (rys. 1.5).

Wizualizacja oprogramowania w istotny sposób korzystająca z techniki animacyjnej jest nazywana **animacją oprogramowania**.



Rys. 1.5. Zasada konstrukcji animacji z poszczególnych ramek

Animacja algorytmów jest kombinacją animacji oprogramowania i wizualizacji algorytmów. Polega na uwidocznieniu wykonanego na wysokim poziomie opisu algorytmu (jego danych, operacji i semantyki) i utworzeniu dynamicznego obrazu, przy zastosowaniu techniki animacji komputerowej.

1.2.5 SYSTEMY WIZUALIZACJI OPROGRAMOWANIA

W rozwiązaniach praktycznych wizualizację oprogramowania najczęściej realizuje się z pomocą specjalizowanego oprogramowania komputerowego, zwanego **systemem wizualizacji oprogramowania**. Odmianą oprogramowania tego typu, którą się będziemy zajmować w niniejszej pracy, są **systemy animacji algorytmów**.

Proces przedstawiania wizualizacji użytkownikowi nazywamy **projekcją**. Zespół elementów sterujących przebiegiem projekcji nazywamy **maszyną projekcyjną**.

Rozróżniamy trzy rodzaje użytkowników systemów wizualizacji oprogramowania. **Programistą** nazywamy osobę, która stworzyła lub dostarczyła program przeznaczony do wizualizacji. Nie musi on wiedzieć, że program ten kiedykolwiek poddany będzie wizualizacji. **Wizualizatorem** (lub, odpowiednio, **animatorem**) nazywamy osobę przygotowującą wizualizację (animację) dostarczonego przez programistę programu, przy wykorzystaniu systemu wizualizacji. **Obserwator** to użytkownik wykorzystujący system do obserwacji projekcji. Jedna i ta sama osoba może grać więcej niż jedną rolę.

1.2.6 WIZUALIZACJA NAUKOWA I PRZEMYSŁOWA

Wizualizacja naukowa ma na celu uwidocznienie określonych cech danych, nie zaś oprogramowania, które je wytwarza. Dane te można interpretować jako model określonego zjawiska fizycznego. Określamy je mianem **danych naukowych**. Mogą one być zarówno efektem symulacji – a zatem wykonania algorytmu modelującego określone zjawisko fizyczne, jak również być wynikiem pomiarów.

Ze względu na znaczną liczbę danych częstokroć generowanych przez oprogramowanie naukowe, w danej chwili można na ogół objąć i zrozumieć tylko niewielką część tych danych. Celem wizualizacji naukowej jest zamiana tego zalewu informacyjnego na kolorowe obrazy (lub, czasami, dźwięki) tak, by stały się one łatwiej przyswajalne. Do typowych obszarów zastosowań należą: obrazowanie danych satelitarnych, symulacja cyfrowa, metody numeryczne, tomografia, meteorologia, klimatologia, geologia, oceanografia itd. Wyczerpujący przegląd dyscypliny można znaleźć w [36].

Przedmiotem **wizualizacji przemysłowej** są obiekty i procesy przemysłowe, w szczególności sterowane komputerowo. Głównym celem tego typu systemów jest wizualizacja wyników pomiarów telemetrycznych, kontrola parametrów procesów produkcyjnych, informowanie o akcjach podjętych przez sterowniki przemysłowe, a także udostępnienie operatorowi systemu możliwości sterowania ręcznego. Systemy wizualizacji przemysłowej interpretują stan sterowanego i wizualizowanego obiektu lub procesu, wykrywając sytuacje specjalne, takie jak awarie i alarmy. Obszerny przegląd zagadnień związanych z wizualizacją przemysłową i systemami wizualizacji przemysłowej można znaleźć w [37, 38].

1.2.7 PROGRAMOWANIE WIZUALNE

Programowanie wizualne jest opisywane często łącznie z wizualizacją, jednak jest ono koncepcją całkowicie odmienną. Podczas gdy w wizualizacji program jest najczęściej specyfikowany w tradycyjny, tekstowy sposób, zaś obraz jest generowany dla zilustrowania jego wybranych cech, to w programowaniu wizualnym obraz graficzny jest stosowany do stworzenia samego programu. Punktem stycznym jest specyfikacja programu stworzonego wizualnie, która jest równocześnie formą wizualizacji tego programu. Przegląd problematyki można znaleźć w [3, 39, 41, 43]. Wizualizację programów specyfikowanych w sposób wizualny opisano w [42, 116, 118].

1.3 TEZA PRACY

Głównym osiągnięciem pracy jest opracowany oryginalny model animacji algorytmów oparty na **śledzeniu przepływu danych**. Model ten został zrealizowany w systemie animacji algorytmów o nazwie *Daphnis*.

Zaproponowana metoda wizualizacji algorytmów jest krokiem w kierunku automatycznego tworzenia wizualizacji algorytmów charakteryzujących się wysokim poziomem abstrakcji. Automatyzacja i wysoki poziom abstrakcji są przez wielu autorów uważane za wymagania sprzeczne [9 – 12], nie dające się pogodzić w pojedynczym rozwiązaniu: im bardziej abstrakcyjna miałaby być wizualizacja, tym więcej wysiłku musi użytkownik włożyć w jej przygotowanie, i *vice versa*, w im większym stopniu zautomatyzowana jest jego praca, tym mniej abstrakcyjna (i atrakcyjna) będzie wizualizacja. Zaproponowana metoda jest środkiem do częściowego pogodzenia tych sprzecznych postulatów. Możemy wykazać, że pozwala ona na wprowadzenie do wizualizacji elementów znacznie **podnoszących poziom abstrakcji bez dodatkowego wysiłku** ze strony wizualizatora. W dotychczas zrealizowanych systemach, przynajmniej tych, które znamy, takie wizualizacje wymagały znacznego wysiłku związanego z precyzyjnym, ręcznym projektowaniem i dostrajaniem wizualizacji. Oczywiście, dla osiągnięcia jeszcze bardziej efektownych rezultatów zaproponowane rozwiązanie może nie być w pełni wystarczające.

Sformułujemy teraz główną tezę pracy:

> Przy zastosowaniu metody animacji algorytmów opartej na śledzeniu przepływu danych możliwe jest automatyczne wprowadzenie do wizualizacji oprogramowania elementów znacząco zwiększających poziom abstrakcji tej wizualizacji

Oryginalnym osiągnięciem pracy jest także propozycja **obiektowej architektury tworzącej łańcuch przekształceń**, przez który przechodzą dane pobrane z poddawanego wizualizacji programu zanim zostaną przedstawione w postaci graficznej. Architektura ta, wykorzystująca koncepcję modelu *POST* [31, 32, 116, 120], stwarza możliwość łatwego konfigurowania ciągu przekształceń danych za pomocą tekstowego pliku skryptowego lub też narzędzi interaktywnych. Stopień komplikacji możliwych w ten sposób do uzyskania przekształceń danych jest tak wysoki, że przy zastosowaniu innych systemów wymaga zwykle zaszycia instrukcji przekształcających dane wprost w tekście źródłowym programu. Wizualizacje o bar-

dzo wysokim stopniu abstrakcji, charakteryzujące się znaczną zawartością intencyjną, mogą być przygotowane w sposób **deklaratywny**. Poprzez zastosowanie różnych wersji skryptu konfiguracyjnego można uzyskiwać różne schematy wizualizacji, o różnych poziomach abstrakcji i szczegółowości – bez potrzeby powtórnej kompilacji programu.

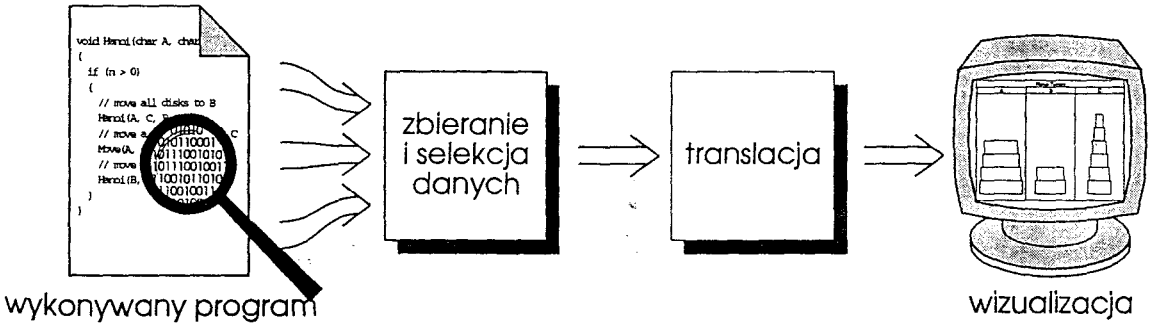
Spośród mniej ważnych, niemniej oryginalnych przyczynków pracy, wymienimy propozycję organizacji powiązania systemu wizualizacji i wizualizowanego programu pozwalającą na uniezależnienie systemu od języka programowania, w którym sformułowano algorytm, oraz otwartą architekturę systemu, która – dzięki zastosowaniu technologii *ActiveX* – pozwala na jego rozszerzanie o moduły zdefiniowane przez użytkownika.

2
PODSTAWY WIZUALIZACJI OPROGRAMOWANIA

Przy całej różnorodności rozwiązań w zakresie wizualizacji oprogramowania, pewien ogólny schemat organizacyjny jest wspólny dla wszystkich systemów (rys. 2.1). Obejmuje on trzy warstwy:

- Zbieranie i selekcja danych. Dane sterujące całą prezentacją muszą być pobrane z oprogramowania poddawanego wizualizacji.
- Translacja danych. Surowe dane nie nadają się do bezpośredniego sterowania końcowym obrazem, muszą zatem być poddane translacji, a przynajmniej skalowaniu.
- Uwidocznienie danych. Przetworzone dane uzyskują swoją końcową formę, niekoniecznie graficzną (por. rozdział 1.2.2).

Tak uproszczone podejście wydaje się sugerować, że strumień danych wejściowych do systemu ogranicza się jedynie do danych ekstrahowanych z poddawanego wizualizacji programu, a danych wyjściowych – do obrazu pokazywanego podczas projekcji (lub też innych



Rys. 2.1. Uproszczona architektura systemu wizualizacji oprogramowania

mediów). W istocie projekcja nie jest możliwa bez wprowadzenia do systemu dostarczanej przez użytkownika specyfikacji, określającej parametry pracy maszyny projekcyjnej. Dodatkowy strumień danych wejściowych jest podczas projekcji dostarczany przez użytkownika, który wchodzi w interakcję z systemem, sterując takimi parametrami projekcji, jak tempo pokazu czy skala obrazu.

Rodzaj i zakres danych wejściowych i wyjściowych systemu zależą od użytkownika. Rozróżniamy trzy grupy użytkowników:

- **Programiści** przygotowują programy przeznaczone do wizualizacji. Dzieje się to w środowisku programowania (ang. *program development environment*), poza systemem wizualizacji (chyba, że oba systemy są zintegrowane).
- **Wizualizatorzy** otrzymują od programisty program i dostarczają go do systemu wizualizacji. Wraz z nim wprowadzają też przygotowaną przez siebie specyfikację wizualizacji. W procesie projektowania wizualizacji powtarzają oni testowe projekcje w sposób iteracyjny, stopniowo udoskonalając specyfikację wizualizacji. Efektem ich pracy jest ostateczna wersja specyfikacji.
- **Obserwatorzy** wprowadzają do systemu program otrzymany od programisty oraz ostateczną wersję specyfikacji, otrzymaną od wizualizatora.

Środki dostępne w typowym systemie wizualizacji zapewniają, że projekcje standardowej jakości mogą być przygotowane stosunkowo łatwo. Do uzyskania wizualizacji wyższej jakości, relatywnie więcej wysiłku trzeba włożyć w jej przygotowanie.

Jakość (czytelność, zrozumiałość, przejrzystość) wizualizacji jest bardzo istotna dla użytkownika obserwującego projekcje. Mają na nią wpływ przede wszystkim dwa czynniki. Pierwszym z nich jest możliwość tworzenia projekcji charakteryzujących się wysokim poziomem **abstrakcji** [6, 7, 10, 12]. Takie projekcje wychodzą poza izomorficzne odwzorowanie kodu czy danych w grafikę. Do zilustrowania semantyki algorytmu niezbędne jest pokazanie nie tylko tych elementów, które wynikają ze stanu programu w danej chwili, ale również takich, które być może w ogóle nie są reprezentowane na poziomie algorytmu. Drugim czynnikiem jest **repertuar reprezentacji graficznych** oferowanych przez system: od prostych technik, takich jak podświetlanie tekstu, aż do zaawansowanej, trójwymiarowej grafiki, animacji i innych efektów multimedialnych [19].

Na łatwość przygotowania projekcji wpływ mają dalsze dwa czynniki. Pierwszym z nich jest **metoda wizualizacji** zastosowana w systemie. Rzutuje ona na rodzaj i sposób przygoto-

wania specyfikacji wizualizacji, a także na charakter i liczbę niezbędnych modyfikacji tekstu źródłowego programu. Drugim czynnikiem jest **wsparcie** oferowane przez system dla użytkownika przygotowującego wizualizację. Czynnik ten jest w literaturze określany często jako **poziom automatyzacji**, chociaż pojęcie wsparcia wydaje się bardziej uzasadnione, ponieważ niektóre środki oferowane przez system nie są w żadnym stopniu automatyczne.

Czynnikiem o kapitalnym znaczeniu dla wszystkich grup użytkowników jest **zakres aspektów** oprogramowania, który może być uwidoczniony przez system podczas wizualizacji. Nie można też pominąć **interfejsu użytkownika**, decydującego o komforcie pracy.

Obszerny przegląd zagadnień związanych z wizualizacją oprogramowania można znaleźć w opublikowanej ostatnio monografii [11], a także w pracach [1 – 12, 74, 116]. Przegląd bibliograficzny dostępny jest w [13].

2.1 ZAKRES WIZUALIZACJI

Systemy wizualizacji oprogramowania koncentrują się na różnych aspektach poddawanego wizualizacji oprogramowania. W zależności od tego, czy wizualizowane aspekty są bliższe szczegółom implementacyjnym programu, czy też abstrakcyjnej metodzie algorytmicznej, rozróżniamy wizualizację programów i wizualizację algorytmów jako dwie wzajemnie się uzupełniające dziedziny wizualizacji oprogramowania [12].

W najprostszym przypadku, wizualizacja koncentruje się na **tekście źródłowym** programu. Debuggery często wyświetlają tekst programu, podświetlając kolejno wykonywane linie. Niektóre systemy stosują techniki typograficzne do ładnego drukowania tekstów programów (ang. *pretty-printing*). Do szeroko stosowanych metod wizualizacji struktur sterowania w programach, wyprowadzanych wprost z tekstu źródłowego, należą diagramy przepływu sterowania (ang. *flowcharts*) oraz diagramy Nassi-Schneidermana [191].

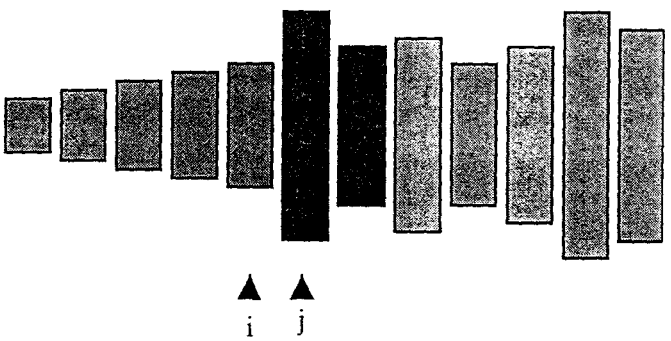
Innym uwidacznianym aspektem oprogramowania są **struktury danych**. Nowoczesne debuggery często zawierają narzędzia do śledzenia danych w formie tekstowej albo semigraficznej. Niektóre systemy wyświetlają graficzną reprezentację struktur zdefiniowanych przez użytkownika.

Wizualizacja **stanu programu w czasie wykonania** (ang. *run-time program state*) obejmuje między innymi wyświetlanie drzewa wywołań (ang. *call graph*), widoku stosu maszynowego (ang. *run-time stack view*), struktury programu, obciążenia systemu, a także tablicy symboli, struktury typów danych i wyrażeń i inne. Niektóre systemy obsługują języki nieim-

peratywne. Możliwe przedmioty wizualizacji to struktury listowe i łańcuchy wywołań funkcji w Lispie, lub też struktura klauzul i celów w Prologu.

Wizualizację wszystkich wymienionych dotąd aspektów oprogramowania można zaliczyć do wizualizacji programów. Wizualizacja **algorytmów** uwidocznia wysokopoziomową strategię programów: celowe działania oraz fundamentalne metody stosowane do rozwiązania konkretnego problemu. Wizualizacja algorytmów nie jest związana z żadną konkretną implementacją, ilustruje zagadnienia z punktu widzenia semantyki, w sposób odmienny od technicznie prostszej wizualizacji struktur danych.

Klasycznym przykładem wizualizacji algorytmów jest graficzne przedstawienie sortowania bąbelkowego (rys. 2.2). Obraz może przedstawiać elementy sortowanej tablicy w postaci prostokątów, których wysokość odpowiada wartościom zmiennych; prostokąty odpowiadające elementom porównywanym w danym kroku algorytmu mogą być podświetlane. Taka właśnie koncepcja graficzna odwzorowuje semantykę algorytmu na obraz graficzny. Taką wizualizację można wzbogacić o odpowiednio zsynchronizowany widok tekstu źródłowego, z zaznaczeniem aktualnie wykonywanej linijki, a także o rozmaite widoki stanu programu, jak drzewa wywołań czy widoku stosu maszynowego. Tak więc, w ramach pojedynczej sesji system wizualizacji może prezentować rozmaite aspekty oprogramowania, w zależności od potrzeb i wymagań użytkownika.



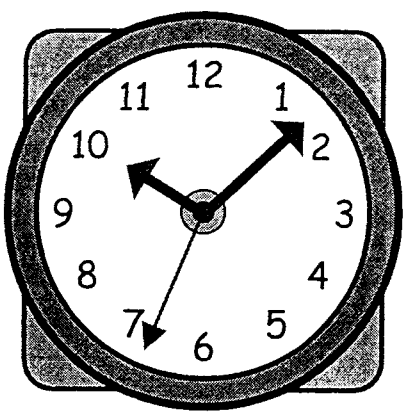
Rys. 2.2. Animacja sortowania bąbelkowego
jako przykład wizualizacji algorytmów

2.2 ABSTRAKCJA

Wysoki poziom abstrakcji jest nieodłączną cechą wizualizacji algorytmów [6, 7, 10, 12]. Metody algorytmiczne, celowe działania, fundamentalne taktyki – wszystko to, co możemy określić jako algorytmiczny aspekt oprogramowania – nie jest *a priori* obecne w programie, jak też nie może być łatwo wywiedzione ani z tekstu programu, ani z jego danych.

O abstrakcji w konkretnej wizualizacji algorytmów świadczą następujące trzy cechy:

- **Abstrahowanie** od tych elementów programu lub algorytmu, które nie są istotne dla zrozumienia algorytmu. Mogą nimi być między innymi pomocnicze struktury danych, wprowadzone w konkretnej implementacji. Dobrym przykładem jest operacja zamiany wartości dwóch zmiennych. Obejmuje ona trzecią, pomocniczą zmienną, której obecność może być uznana za nieistotną i tym samym zignorowana.
- **Zmiana realizacji** algorytmu. Czasami faktycznie zastosowana implementacja nie sprzyja zrozumieniu algorytmu. Przykładem niech będzie algorytm symulacji automatu komórkowego, realizowany na konwencjonalnej maszynie asynchronicznej. Dla potrzeb wizualizacji można przedstawić realizację synchroniczną, bliższą teoretycznemu modelowi automatu, chociaż niemożliwą do zaimplementowania na zwykłym komputerze.
- Dodanie **zawartości intencyjnej**, to jest elementów nie występujących (*a priori*) w programie poddawanym wizualizacji. Najłatwiejszym sposobem jest dobór odpowiednich reprezentacji graficznych, stanowiących trafne metafory przedstawianych danych. Na przykład trzy zmienne całkowite o nazwach *godzina*, *minuta* i *sekunda* można przedstawić w postaci tarczy zegara z godzinową, minutową i sekundową wskazówką (rys. 2.3). Bardziej rozwinięte wizualizacje mogą pokazywać struktury danych nie występujące w rzeczywistym algorytmie. Na przykład wizualizacja algorytmu tablicy mieszającej może za-



Rys. 2.3. Przykład wizualizacji z zawartością intencyjną: ilustracja trzech zmiennych całkowitych

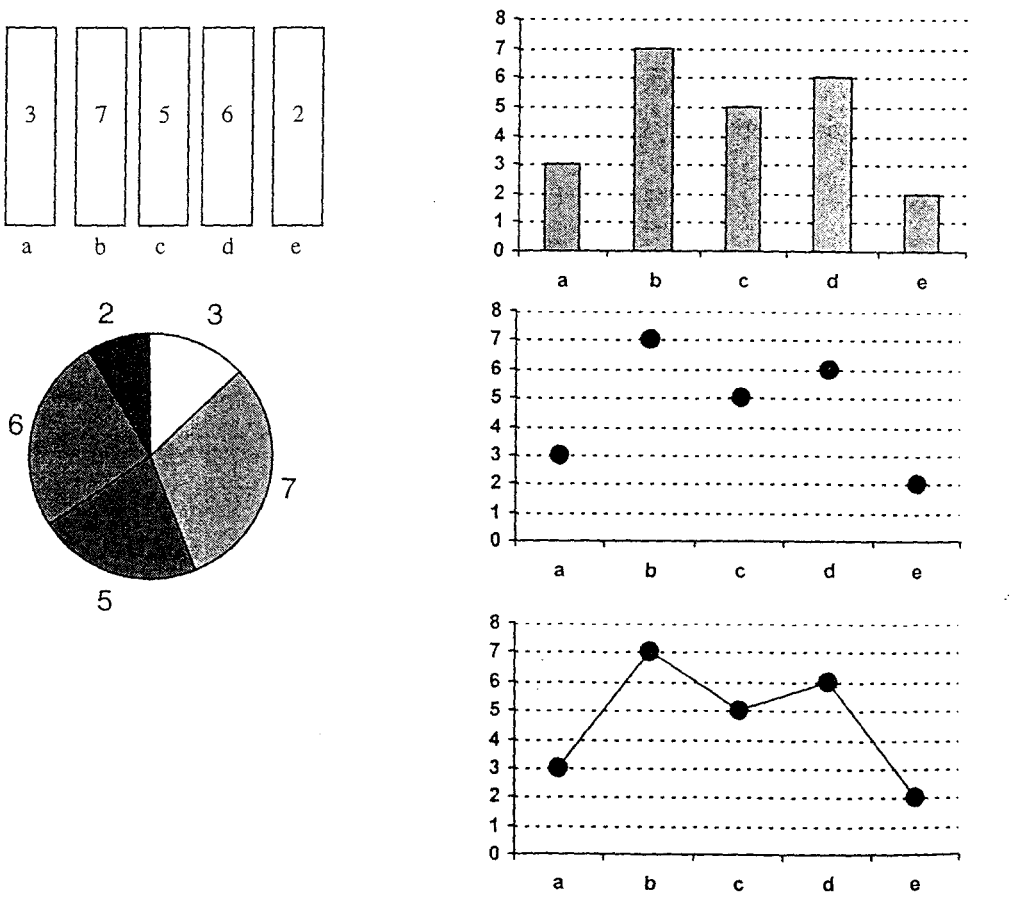
wierać widok struktury opisującej statystykę konfliktów występujących podczas dostępu do tablicy (por. plansze 8.5 i 8.6).

2.3 REPERTUAR GRAFICZNY

Nie mający precedensu rozwój grafiki komputerowej i animacji w ostatnich latach zaowocował systemami wizualizacji o coraz bogatszym repertuarze środków graficznych [19]. Wiele z nich obsługuje animowane, trójwymiarowe projekcje [18, 21 – 23]. Szeroki wybór narzędzi graficznych pozwala wizualizatorowi na wybór adekwatnej reprezentacji danych, odznaczającej się wysokim wskaźnikiem zawartości intencyjnej.

Zazwyczaj poddawane wizualizacji wielkości mają wiele potencjalnie możliwych reprezentacji graficznych. Na przykład tablica liczb całkowitych może być przedstawiona w klasycznym formacie „pudełkowym", w postaci wykresu słupkowego, kołowego, punktowego czy liniowego (rys. 2.4). W przypadku konkretnego algorytmu pewne reprezentacje są bardziej adekwatne niż inne, na przykład dla algorytmu sortowania właściwa jest reprezentacja w postaci wykresu słupkowego albo punktowego, podczas gdy wykres kołowy byłby zupełnie niewłaściwy.

Mimo iż bogaty repertuar środków graficznych jest bardzo istotny przy projektowaniu wizualizacji, w praktyce jednak tylko ograniczony podzbiór reprezentacji jest szeroko wykorzy-



Rys. 2.4. Różne reprezentacje graficzne pięcioelementowej tablicy liczb

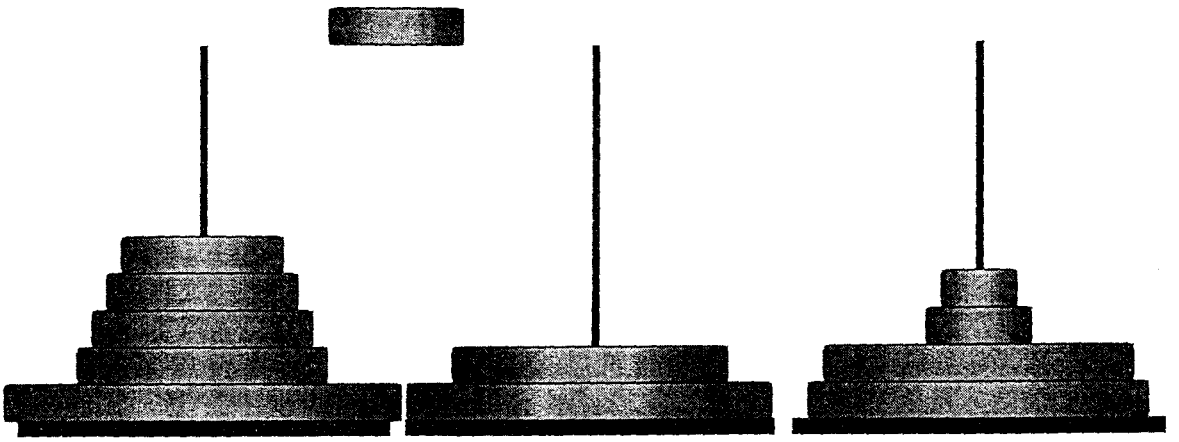
stywany w wizualizacjach [122]. Wiele zaawansowanych technik graficznych znajduje stosunkowo wąskie zastosowania. Na przykład prezentacje trójwymiarowe są idealnym rozwiązaniem przy przedstawianiu wielowymiarowych danych, ale stanowią jedynie element dekoracyjny w większości innych zastosowań. Jednak w prawie każdym przypadku techniką o fundamentalnym znaczeniu jest animacja [12].

Animacja wprowadza do wizualizacji niezwykle cenny ładunek zawartości intencyjnej, poprzez ukazanie stanów pośrednich pomiędzy konfiguracjami faktycznymi. Za konfigurację faktyczną (ang. *valid configuration*) uważamy stan programu mający określoną semantykę, i który jest osiągalny podczas procesu realizacji programu. Gdy wykonywany jest program, przechodzi on od jednej konfiguracji faktycznej do następnej. Reprezentują one fazy algorytmu, sensowne w odniesieniu do jego celu, na przykład po wykonaniu każdej instrukcji albo po ukończeniu każdej operacji zamiany w algorytmie sortowania. Jednak jednym z głównych celów wizualizacji algorytmów jest nie tylko pokazanie konfiguracji faktycznych, które program osiąga w trakcie wykonania, ale także przejść pomiędzy nimi. Tak więc system animacji pokazuje nie tylko obrazy odnoszące się do konfiguracji faktycznych, ale także stany pośrednie pomiędzy nimi. Animacja pozwala na uzyskanie iluzji ciągłego, płynnego ruchu pomiędzy kolejnymi konfiguracjami. Stany pośrednie, osiągane w trakcie tego ruchu, nie mają semantycznego odpowiednika w wykonywanym programie, ale ułatwiają one użytkownikowi zrozumienie semantyki.

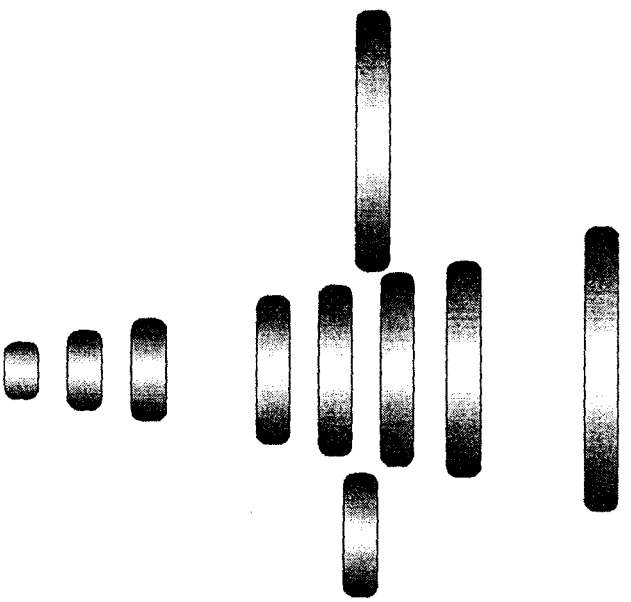
Rozpatrzmy przykład animacji algorytmu wież Hanoi (rys. 2.5). Przedstawienie krążków w położeniach pośrednich, przesuwających się pomiędzy palikami, jest krytycznym czynnikiem dla zrozumienia działania algorytmu, choć nie znajduje odniesienia do jakiejkolwiek konfiguracji faktycznej algorytmu. Zwykłe przedstawienie sekwencji kadrów przedstawiających krążki w ich końcowych położeniach byłoby zupełnie nieczytelne. Innym dobrym przykładem są algorytmy sortowania (klasyczny przykład animacji algorytmów). Skokowe zmiany wartości elementów byłyby bardzo trudne do prześledzenia, w porównaniu z animacją, w której elementy płynnie przesuwają się zmieniając swoje pozycje (rys. 2.6).

2.4 INTERFEJS UŻYTKOWNIKA

Systemy wizualizacji oprogramowania, podobnie jak wszystkie nowoczesne produkty programowe, muszą być podporządkowane ogólnym zasadom dotyczącym projektowania interfejsu użytkownika [184, 187, 190, 192]. Do najważniejszych postulatów w tej dziedzinie



Rys. 2.5. Animacja algorytmu „Wieże Hanoi"



Rys. 2.6. Animacja algorytmu sortowania

należy wymóg, by systemy były przyjazne dla użytkownika i dobrze dopasowane do jego potrzeb, wymagań i kompetencji.

W przypadku wizualizacji oprogramowania mamy do czynienia z dwiema głównymi grupami użytkowników: obserwatorami i wizualizatorami (programistów w tym miejscu pomijamy, gdyż nie wchodzą oni w bezpośrednią interakcję z systemem). Obserwator jest zasadniczo użytkownikiem pasywnym. Do najważniejszych jego wymagań należy czytelność i zrozumiałość wizualizacji. Wpływ na to mają przede wszystkim czynniki opisane w poprzednich punktach: zakres wizualizacji, poziom abstrakcji i repertuar graficzny. Oprócz tego, powinien

on móc z łatwością uruchomić żądaną wizualizację. W czasie projekcji, do dyspozycji obserwatora pozostają elementy interfejsu odnoszące się do następujących sfer interakcji z systemem:

- Interakcja na poziomie danych, która pozwala użytkownikowi na dynamiczną zmianę zawartości obserwowanych struktur danych podczas trwania projekcji.
- Kontrola skali, która pozwala użytkownikowi na określenie szczegółowości, czy choćby tylko rozmiaru prezentowanego obrazu. Znajduje zastosowanie przede wszystkim przy dużych problemach, których całościowa prezentacja nie byłaby możliwa. Wiele systemów pozwala na wybór interesujących wątków projekcji, umożliwiając skoncentrowanie się jedynie na wybranych aspektach wykonania algorytmu.
- Kontrola tempa, pozwalająca użytkownikowi sterować szybkością projekcji, przerywać ją, a w niektórych systemach nawet cofać lub odtwarzać w odwróconej kolejności.

Wymienione czynności często określa się mianem **reżyserowania projekcji** [26].

Co do wizualizatora, udostępniony mu interfejs użytkownika w głównej mierze zależy od architektury systemu i przyjętej metody wizualizacji, i nie może być opisany w sposób uogólniony. Systemy często oferują różnego typu wsparcie dla użytkowników przygotowujących wizualizację. Problematyka ta zostanie naświetlona w dwóch następnych punktach.

2.5 METODA WIZUALIZACJI

Metoda wizualizacji jest jedną z najistotniejszych cech systemów wizualizacji oprogramowania, szczególnie z punktu widzenia wizualizatora. Obejmuje ona dwa główne aspekty:

- **Metoda specyfikacji**: decyduje o sposobie zdefiniowania i skonfigurowania wizualizacji.
- **Metoda połączenia**: decyduje o sposobie połączenia poddawanego wizualizacji programu z systemem.

Wizualizacja może być całkowicie **kodowana ręcznie**: wizualizator pisze specjalny program, którego celem jest zobrazowanie działania określonego algorytmu lub innego programu, być może przy użyciu dedykowanej biblioteki graficznej. Taki program jest sam przez się specyfikacją wizualizacji; nie ma w tym przypadku problemu metody połączenia.

Bardziej zaawansowaną metodą jest **wizualizacja sterowana zdarzeniami** (ang. *event-driven visualisation*). Wizualizator uzupełnia tekst źródłowy programu przeznaczonego do wizualizacji. Każde uzupełnienie (ang. *annotation*) odnosi się do wystąpienia tzw. **zdarzenia interesującego** (ang. *interesting event*): określonego zdarzenia, które powinno zostać uwidocznione przez system, przy czym każdorazowo podaje się parametry definiujące reprezen-

tację graficzną tego zdarzenia. Uzupełnienia tekstu źródłowego stanowią więc specyfikację wizualizacji. Typ połączenia między programem a systemem jest typowo **inwazyjny**, to znaczy wymaga zmodyfikowania wizualizowanego oprogramowania. Metoda specyfikacji może zostać ulepszona przez użycie skryptów: dodatkowych plików definiujących reprezentacje graficzne dla różnych typów zdarzeń. Wówczas uzupełnienia tekstu źródłowego zawierają jedynie odniesienia do odpowiednich pozycji w obrębie skryptu. Niektóre systemy idą nawet dalej i oferują specjalne edytory lub preprocesory tekstu, wspomagające proces uzupełniania tekstu. We wszystkich przypadkach mówimy jednak, że metoda specyfikacji pozostaje **imperatywna**: każde interesujące zdarzenie (czyli *de facto* miejsce w tekście źródłowym) jest związane z określoną *a priori* operacją graficzną.

W przypadku **wizualizacji sterowanej danymi** (ang. *data-driven visualisation*) jedynie fakt zmiany wartości danych jest anonsowany systemowi. Znacznie zmniejsza to różnorodność możliwych uzupełnień tekstu źródłowego. W niektórych systemach stosuje się preprocesory, podczas gdy inne zawierają odpowiednio spreparowane środowisko wykonania (interpretatory lub kompilatory), tak, że struktury danych mogą być monitorowane bez potrzeby jakiegokolwiek modyfikowania tekstu źródłowego. Ta ostatnia technika połączenia programu z systemem, zwana połączeniem **nieinwazyjnym**, jest szczególnie dobrze dostosowana do wielu systemów pracujących w środowisku języka Smalltalk: standardowe cechy tego języka i jego środowiska są wystarczające do zapewnienia w pełni automatycznego powiązania pomiędzy programem a systemem wizualizacji. Styl specyfikacji wizualizacji w systemach sterowanych danymi jest **deklaratywny**: specyfikuje się, w jaki sposób obrazować określoną strukturę danych, bez względu na to, kiedy (w której linii programu) jej zawartość zostanie zmieniona i w jaki sposób. Specyfikacja ma często formę skryptu – tekstowego pliku konfiguracyjnego.

W większości przypadków projekcja odbywa się równocześnie z wykonaniem przedstawianego programu. W niektórych systemach stosuje się jednak tzw. wizualizację **post-mortem**: cały pokaz jest przedstawiany po zakończeniu wykonywania programu, na podstawie zarejestrowanego śladu wykonania programu (ang. *program trace*). Ma to znaczenie przede wszystkim przy wizualizowaniu programów pracujących w ograniczeniach czasu rzeczywistego: narzut związany ze sterowaniem wizualizacją mógłby opóźnić działanie programu w sposób nieakceptowalny. Z tego samego powodu styl połączenia takich systemów jest bardzo często nieinwazyjny. Wizualizacja post-mortem może też znaleźć zastosowanie w przypadku systemów realizujących obliczenia w ciągu bardzo długiego czasu.

2.6 WSPARCIE ZE STRONY SYSTEMU I AUTOMATYZACJA

Środki wspomagające i automatyzujące pracę wizualizatora w dużym stopniu zależą od rodzaju systemu.

Widoki struktur danych w prostych systemach wizualizacji programów są często generowane automatycznie, bez udziału użytkownika. Debuggery monitorują stan programu w danym momencie i odczytują dane niezbędne do wygenerowania prostych widoków danych. Możliwość taka, bez konieczności przeznaczania czasu na projektowanie wizualizacji, jest niezbędna przy realizacji czasochłonnych zadań, takich jak uruchamianie programów. Bardziej rozbudowane systemy pozwalają użytkownikowi na pewną modyfikację wygenerowanego obrazu.

Wizualizacja kodu programu, a także stanu w czasie wykonania (np. drzewo wywołań, widoki stosu maszynowego itd.) również może być relatywnie łatwo w pełni zautomatyzowana. Najtrudniejszym elementem projektowania takich widoków nie jest generowanie grafiki, ale raczej akwizycja danych, wymagająca zwykle programowania niskopoziomowego dla uzyskania dostępu do wewnętrznych struktur programu, tablic symboli itd.

W przeciwieństwie do opisanych powyżej, systemy wizualizacji algorytmów wyświetlają informacje, które częstokroć nie mogą być w sposób automatyczny wyprowadzone ze stanów programu podczas procesu jego wykonania. Chodzi oczywiście o pojęcia semantyczne i abstrakcje użyte w algorytmie. Wizualizacja algorytmów wymaga stosowania widoków charakteryzujących się wysokim poziomem abstrakcji, ze znaczną zawartością intencyjną. Według wielu autorów [9 – 12, 73] istnieje głęboka sprzeczność pomiędzy postulatem zwiększania poziomu abstrakcji z jednej strony, a potrzebą automatyzacji procesu przygotowania wizualizacji z drugiej. Jednocześnie inni autorzy twierdzą, że do systemów wizualizacji oprogramowania trzeba jeszcze wprowadzić znacznie więcej automatyzacji, jeśli kiedykolwiek mają się one stać użytecznym narzędziem w rękach inżyniera. W przeciwnym razie, wysiłek włożony w przygotowanie animacji przewyższy spodziewany efekt pozytywny. Jednak automatyczne systemy nie są wystarczające, gdyż tworzone przy ich użyciu wizualizacje są zbyt prymitywne. Pożądanym rozwiązaniem byłby automatyczny system rozumiejący semantykę programów, który pozwalałby na tworzenie zaawansowanych wizualizacji nie wymagając od użytkownika znajomości poddawanego wizualizacji kodu [5].

Istniejące systemy, choć nie automatyczne, oferują szereg udogodnień ułatwiających pracę wizualizatora i skracających cykl prac związanych z przygotowaniem animacji. Obejmują one: preprocesory tekstu, które automatycznie wprowadzają uzupełnienia do tekstów źródło-

wych programów, narzędzia skryptowe i interaktywne do projektowania widoków graficznych i wreszcie edytory do tworzenia elementów graficznych. Coraz więcej systemów oferuje też możliwości specyfikowania wizualizacji metodami wizualnymi, za pomocą bezpośredniej manipulacji elementami graficznymi [33 – 35, 79, 101, 102].

2.7 ZASTOSOWANIA WIZUALIZACJI OPROGRAMOWANIA

Systemy wizualizacji oprogramowania znajdują zastosowania przede wszystkim w dwóch głównych dziedzinach:

1. Dydaktyka informatyki,
2. Projektowanie i uruchamianie oprogramowania.

Bez cienia wątpliwości, dydaktyka jest polem zastosowań nie tylko historycznie wcześniejszym, ale też ogólnie bardziej popularnym, i tu właśnie odnotowano większe sukcesy. Systemy wizualizacji ekscytowały zarówno studentów, jak i nauczycieli. Film Ronalda M. Backera *Sorting Out Sorting* [59], często określany jako prekursor udanej animacji algorytmów, został stworzony jako narzędzie do nauczania algorytmiki. Po prawie 20 latach jest wciąż dostępny w handlu i wciąż chętnie sięgają po niego nauczyciele. Marc H. Brown jako pierwszy użył systemu animacji do nauczania algorytmiki w swojej „elektronicznej klasie", utworzonej we wczesnych latach osiemdziesiątych w Brown University [47 – 50]. Na początku lat dziewięćdziesiątych organizował on festiwale animacji algorytmów [45, 46], które były świetną okazją dla nauczycieli informatyki do zapoznania się z tą technologią. Najistotniejszym problemem była trudność posługiwania się systemami.

Prace nad dydaktycznymi zastosowaniami wizualizacji algorytmów były prowadzone przez grupę kierowaną przez Johna T. Stasko z Georgia Institute of Technology [52, 53, 56 – 58]. Próbował on empirycznie oszacować przydatność animacji w nauczaniu informatyki. Jego wyniki pokazały, że studenci, którzy w toku zajęć laboratoryjnych mogli stworzyć własne animacje wybranych algorytmów, osiągnęli wyraźnie lepsze wyniki w kończącym badania egzaminie sprawdzającym zrozumienie algorytmów, niż grupa studentów, którzy jedynie biernie obserwowali zawczasu przygotowane przykładowe animacje. Nawet jednak ta ostatnia grupa okazała się lepsza od grupy kontrolnej, która w ogóle nie miała styczności z narzędziami wizualizacji algorytmów.

Niemal wszystkie systemy doczekały się zastosowań w dydaktyce. Wiele z nich zostało specjalnie w tym celu zaprojektowanych. Na przykład Peter A. Gloor stworzył system anima-

cyjny Aace [123], i wykorzystał go do stworzenia i opublikowania hipermedialnego krążka CD-ROM pomagającego w nauce algorytmów [124].

W przeciwieństwie do zastosowań dydaktycznych, systemy wizualizacji oprogramowania wciąż z wielkim trudem przyjmują się jako narzędzia inżynierskie, wspomagające projektowanie, uruchamianie i rozwój oprogramowania. Mimo kilku prac dotyczących uruchamiania programów za pomocą wizualnych debuggerów [54, 55], a nawet nielicznych dostępnych komercyjnie systemów projektowania oprogramowania korzystających z technik wizualizacyjnych (na przykład francuski *Centaur* [66, 67]), wciąż wydaje się, że systemy tego typu są zbyt skomplikowane w obsłudze, by stanowić skuteczne i wygodne narzędzie. Z drugiej strony, systemy wizualizacji algorytmów wywarły istotny i niezaprzeczalny wpływ na współczesne środowiska projektowania oprogramowania [44], które zyskują cechy do niedawna jeszcze zarezerwowane wyłącznie dla prototypowych systemów wizualizacji.

Krytyczną analizę przydatności systemów wizualizacji oprogramowania można znaleźć między innymi w [2, 4].

3
AKTUALNY STAN DZIEDZINY

Chociaż dziedzina zwana wizualizacją oprogramowania liczy sobie już ponad 30 lat, jednak za pracę, która zapoczątkowała jej faktyczny rozwój i stanowiła inspirację dla późniejszych rozwiązań uważa się film animowany Ronalda M. Baeckera *Sorting Out Sorting*, pokazany po raz pierwszy w trakcie konferencji SIGGRAPH w 1981 roku. Następne 18 lat wyznacza okres badań i rozwój narzędzi wizualizacji oprogramowania. W początkowym okresie dominowały filmy animowane; w późniejszych latach główny nurt działań przesunął się ku tworzeniu dedykowanych systemów komputerowych. W chwili obecnej rozwiązania z zakresu wyświetlania struktur danych i stanu programu, czy ogólniej, wizualizacji programów, na dobre zagościły już w nowoczesnych środkach projektowania oprogramowania. Wizualizacja algorytmów, która jest naszym głównym przedmiotem zainteresowania, stanowi dobre narzędzie dydaktyczne w informatyce, jednak dopiero zaczyna mieć wpływ na narzędzia inżynierii oprogramowania. Skoncentrujemy się na systemach wizualizacji algorytmów przeznaczonych dla algorytmów zapisanych w klasycznych językach imperatywnych. Dokonamy też krótkiego przeglądu rozwiązań z zakresu wizualizacji nieimperatywnych aspektów oprogramowania.

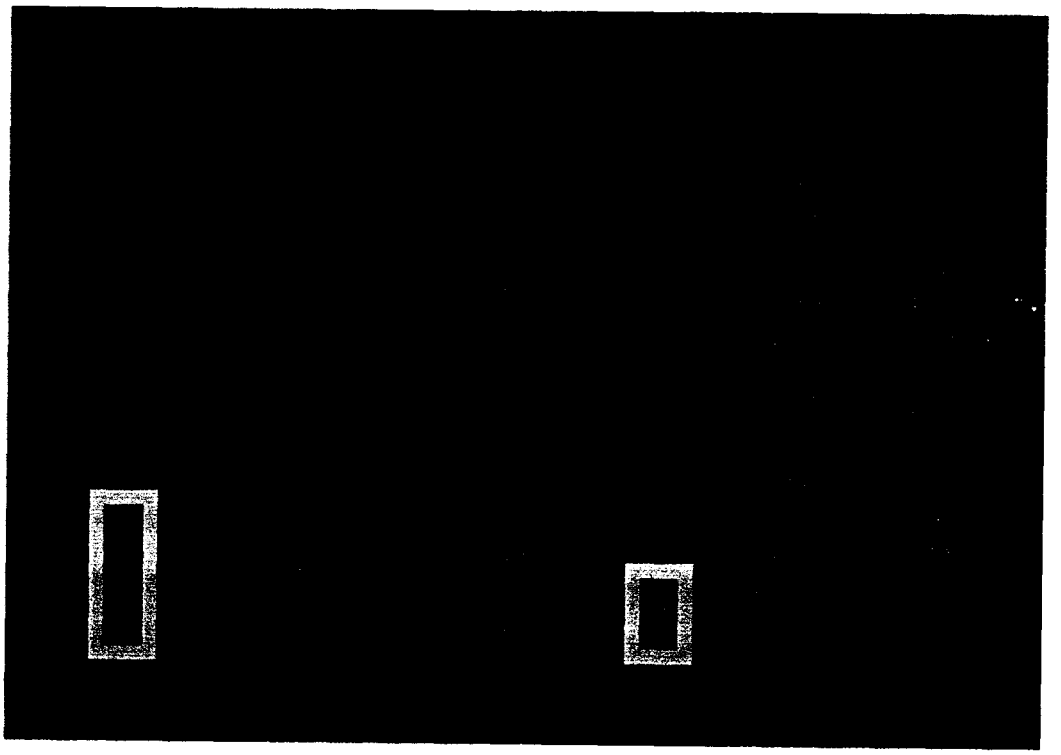
3.1 FILMY ANIMOWANE

Początki wizualizacji oprogramowania sięgają połowy lat sześćdziesiątych. W tamtych czasach największym problemem związanym z tworzeniem systemów intensywnie wykorzystujących techniki graficzne był wysoki koszt stacji graficznych [51, 61]. Przy bardzo niewielkim dostępie do sprzętu, rozwiązanie było oczywiste: ponieważ koszt wykorzystania komputera był znacznie wyższy niż koszt wykonania kopii filmowej, zatem pierwsze wizualizacje rejestrowano na taśmie filmowej i rozpowszechniano w postaci filmów animowanych.

Głównym ograniczeniem filmów animowanych był brak możliwości interaktywnej pracy użytkownika. Widzowie oglądali przebieg projekcji dokładnie w takiej postaci, w jakiej została ona wytworzona. Nie było możliwości eksperymentowania z obrazem. Aż do początku lat osiemdziesiątych stworzono stosunkowo niewiele takich filmów. Interesujące, że w tym czasie powstało znacznie więcej filmów dotyczących innych dziedzin, jak matematyka, fizyka, chemia czy biologia, przy których wytworzeniu korzystano z technik komputerowych. Zaskakująco słabo była wśród nich reprezentowana informatyka. W 1971 roku Huggins stwierdził: „osobliwe, że umiejętność wyrażania się za pomocą środków wizualnych wydaje się zupełnie obca praktykującym w tej dziedzinie" [201]. Niestety jeszcze i dzisiaj, w końcu lat dziewięćdziesiątych, musimy stwierdzić, że wizualizacja oprogramowania mocno ustępuje pod względem popularności innym dziedzinom wizualizacji naukowej czy przemysłowej.

Pierwszy wygenerowany przez komputer film dotyczący tematyki komputerowej został stworzony przez K. C. Knowltona z Bell Labs, w roku 1966. Film *L6: Bell Telephone Laboratories Low-Level Linked List Language* [62, 63] ilustrował *L6*, język przetwarzania list na poziomie asemblera. Pierwszy film, który możemy nazwać *wizualizacją algorytmu*, został stworzony przez Hopgooda [61] w 1974 roku i dotyczył algorytmów mieszających. Przedstawiał on trzy dyskretnie odświeżane widoki: widok tablicy mieszającej, pokazujący także, ile kolizji nastąpiło przed wprowadzeniem nowego elementu, wykres obrazujący liczbę prób potrzebnych do wprowadzenia poszczególnych elementów do tablicy i wreszcie wartość maksymalnej liczby kolizji. W filmie K. S. Bootha *PQ-Trees* [60] z 1975 roku, pokazującym działanie różnych algorytmów na strukturze danych typu drzewo *PQ*, po raz pierwszy zastosowano płynną animację.

Jednak bez wątpienia najlepiej znanym filmem jest *Sorting Out Sorting* [59] Ronalda M. Baeckera. Przedstawia on dziewięć różnych algorytmów sortowania, przy czym każdy z nich zastosowano osobno do małej i dużej próbki danych (plansza 3.1). Małe zbiory danych reprezentowane były jako zestaw prostokątów o wysokości zależnej od wartości danych, podczas gdy duże zestawy przedstawiono w postaci wykresu punktowego. Kropki reprezentujące poszczególne elementy tworzyły całe mgławice, kształtujące się w różny sposób tworząc charakterystyczne dla poszczególnych algorytmów formy. Jedną z najciekawszych scen filmu był wyścig algorytmów: wszystkie algorytmy przedstawiono razem, dając możliwość porównania ich wydajności. Ciekawe, że produkcja tego półgodzinnego, kolorowego filmu zajęła zespo-



Plansza 3.1. Kadr z filmu *Sorting Out Sorting* R. M. Baeckera, ilustrujący algorytm sortowania przez proste wstawianie [11]

łowi jego twórców cztery lata! Dodajmy, że ten prekursorski film jest w dalszym ciągu dostępny na kasetach i wykorzystywany w dydaktyce informatyki.

W latach osiemdziesiątych i dziewięćdziesiątych sprzęt komputerowy stał się relatywnie tani, i filmy zostały zastąpione systemami komputerowymi. Wciąż jednak wiele spośród stworzonych systemów było wykorzystywanych do produkcji taśm wideo, na których rejestrowane były wybrane wizualizacje.

3.2 NARZĘDZIA WIZUALIZACJI PROGRAMÓW

Systemy i narzędzia przeznaczone do wizualizacji programów mają wyraźną przewagę nad większością systemów wizualizacji algorytmów w tym, że pozwalają na bardzo szybkie i łatwe obserwowanie tych aspektów programów, do których zostały przeznaczone, bez żmudnego etapu projektowania charakterystycznego dla środków wizualizacji algorytmów. Mimo ograniczonych możliwości narzędzia wizualizacji programów znalazły komercyjne uznanie, szczególnie w dziedzinie projektowania i uruchamiania programów – czego wciąż nie można jeszcze powiedzieć o systemach wizualizacji algorytmów.

Wymagania stawiane wizualizacji programów w zastosowaniu do uruchamiania programów są zupełnie inne, niż w przypadku wizualizacji algorytmów. Widoki muszą być precyzyjniejsze i bardziej złożone, tak by umożliwiały wykrycie niepoprawnych danych. W szczególności, wizualizowane dane muszą się samoidentyfikować, poprzez wyświetlenie nazwy i dokładnej wartości. W przeciwieństwie do tego, w wizualizacji algorytmów możemy sobie pozwolić na pewne uproszczenia i niedokładności, aby tylko uzyskać pożądany efekt. Z drugiej strony, wizualizacja algorytmów operuje często dodatkowymi elementami, nieobecnymi w wizualizowanym oprogramowaniu, ale ułatwiającymi zrozumienie algorytmu. Elementy takie nie są zazwyczaj obecne w wizualizacji programów. Podsumowując możemy stwierdzić, iż wizualizacja programów charakteryzuje się niższym poziomem abstrakcji, niż wizualizacja algorytmów.

ŁADNE DRUKOWANIE PROGRAMÓW

Jak już wspominaliśmy, jedną z najprostszych metod automatycznej wizualizacji oprogramowania jest ładne drukowanie programów. Prototyp kompilatora wizualnego *SEE* R. M. Baeckera i A. Marcusa [64, 65] jest w pełni zautomatyzowanym systemem wizualizacji, wykorzystującym zaawansowane wyniki badań nad tzw. czynnikiem ludzkim (ang. *human fac-*

tors), oraz wyrafinowane techniki typograficzne w celu sporządzenia wysokiej jakości wydruków tekstów źródłowych programów napisanych w języku *C*.

Twórcy systemu *SEE* myśleli o wydruku programu komputerowego jak o każdym innym tekście technicznym: złożonym z rozdziałów, zaopatrzonym w spis treści oraz indeks. W typowym listingu wygenerowanym przez *SEE* nagłówki funkcji drukowane są wytłuszczoną czcionką, z cienką linią biegnącą w poprzek strony. Parametry i zmienne lokalne[1] drukowane są poniżej w dwóch kolumnach. Do wydruku tekstu programu stosuje się czcionkę bezszeryfową, a do komentarzy – szeryfową. Wielowierszowe komentarze wyróżnione są szarym tłem, natomiast krótsze komentarze są drukowane mniejszą czcionką na marginesie, na lewo od linijki tekstu, której dotyczą.

System *SEE* jest zapewne jednym z najpełniejszych i najpotężniejszych narzędzi do ładnego drukowania programów. Innym interesującym rozwiązaniem jest francuski system Centaur [66, 67], opracowany w *INRIA* (*Institut National de Recherche en Informatique et en Automatique*). Jest to rozbudowane narzędzie przeznaczone do tworzenia rozproszonych środowisk projektowania oprogramowania w oparciu o formalną specyfikację języka programowania. Posiada rozbudowane możliwości projektowania graficznego interfejsu użytkownika. Moduł ładnego drukowania jest tylko jednym z wielu możliwych do stworzenia modułów, takich jak parser, interpretator i kompilator. System został z sukcesem wykorzystany do stworzenia środowisk programowania w językach imperatywnych (*C*), funkcyjnych (*ML*), obiektowych (*Eiffel*), a także języków dedykowanych dla obliczeń równoległych opartych o zasadę przepływu danych (*Sisal*) [68]. System *Centaur* jest dostępny komercyjnie.

System *WEB* D. E. Knutha [69] jest rozszerzeniem języka T_EX. Ułatwia integrowanie sformatowanych listingów w ramach obszerniejszych dokumentów. Z drugiej jednak strony system wymaga ręcznego formatowania programów, gdyż sam nie dokonuje rozbioru gramatycznego formatowanego tekstu.

Rozwiązania podobne do opisanych znalazły szerokie zastosowanie we współczesnych środowiskach projektowania programów, jakkolwiek spotykane rozwiązania najczęściej ograniczają się do wyróżniania kolorami lub wytłuszczeniem określonych części listingów, i są znacznie uboższe niż opisane powyżej narzędzia.

---

[1] System *SEE* obsługuje język *C* w starej składni, z listą parametrów po nagłówku funkcji.

INCENSE, AMETHIST I PASCAL GENIE

Incense, stworzony przez B.A. Myersa z *Xerox PARC* we wczesnych latach osiemdziesiątych, był pierwszym systemem, który automatycznie wyświetlał widoki struktur danych [71]. System pracował w połączeniu z *Mesą*, językiem programowania podobnym do *Pascala*, i przeznaczony był dla doświadczonych programistów. Myers i jego współpracownicy z *Carnegie-Mellon* opracowali następnie system nazwany *Amethyst* [70], zintegrowany ze środowiskiem programowania dla komputerów *Macintosh* o nazwie *MacGnome Pascal*, znanym później jako *Pascal Genie*. System ten został zaprojektowany specjalnie z myślą o studentach i był wyposażony w możliwości automatycznego zarządzania wyświetlaniem, a także oferował efekty animacyjne.

System generował graficzny widok struktur danych wyspecyfikowanych przez użytkownika. Korzystając z tablicy symboli automatycznie tworzył naturalny obraz zmiennych programu. Obraz ten tworzony był na podstawie wbudowanych *formatów*, związanych z każdym typem występującym w języku *Mesa* (a potem *Pascal*).

MICROSOFT VISUAL C++

Komercyjne środowiska programowania stosunkowo wolno przyjmowały nowinki z dziedziny interfejsu użytkownika. Tradycyjne narzędzia – jak edytory listingów, przeglądarki tekstu źródłowego czy debuggery – zostały wzbogacone o elementy interfejsu okienkowego. Jednak wiele nowocześniejszych rozwiązań poszło o krok dalej i stosuje bardziej wyrafinowane środki graficzne do przedstawienia informacji, stając się dość zaawansowanymi systemami wizualizacji programów.

Microsoft Visual C++ [72] jest tylko jednym z wielu przykładów nowoczesnych produktów efektywnie korzystających z technik wizualnych, w tym z pewnych form wizualizacji programów. Inne przykłady obejmują: pozostałych członków rodziny produktów wizualnych firmy *Microsoft* (*Java*, *Basic*, *Fortran*, *FoxPro*), produkty firmy *Borland*: *C++*, *C++ Builder* i paskalopodobny *Delphi*, *WHATCOM C/C++*, *PowerBuilder*, *ParcPlace SPARCworks* i *Visual Smalltalk*, by nie wspomnieć o licznych wizualnych narzędziach projektowania systemów bazodanowych. Jednym z pierwszych produktów handlowych korzystających z technik wizualizacyjnych do wspomagania projektowania programów w języku *C++* był *CenterLine ObjectCenter*. O tym, że właśnie *Visual C++* został wybrany jako reprezentant tego typu

systemów zadecydował między innymi fakt, że właśnie to środowisko zostało wykorzystane do prac implementacyjnych opisywanych w dalszej części niniejszej pracy.

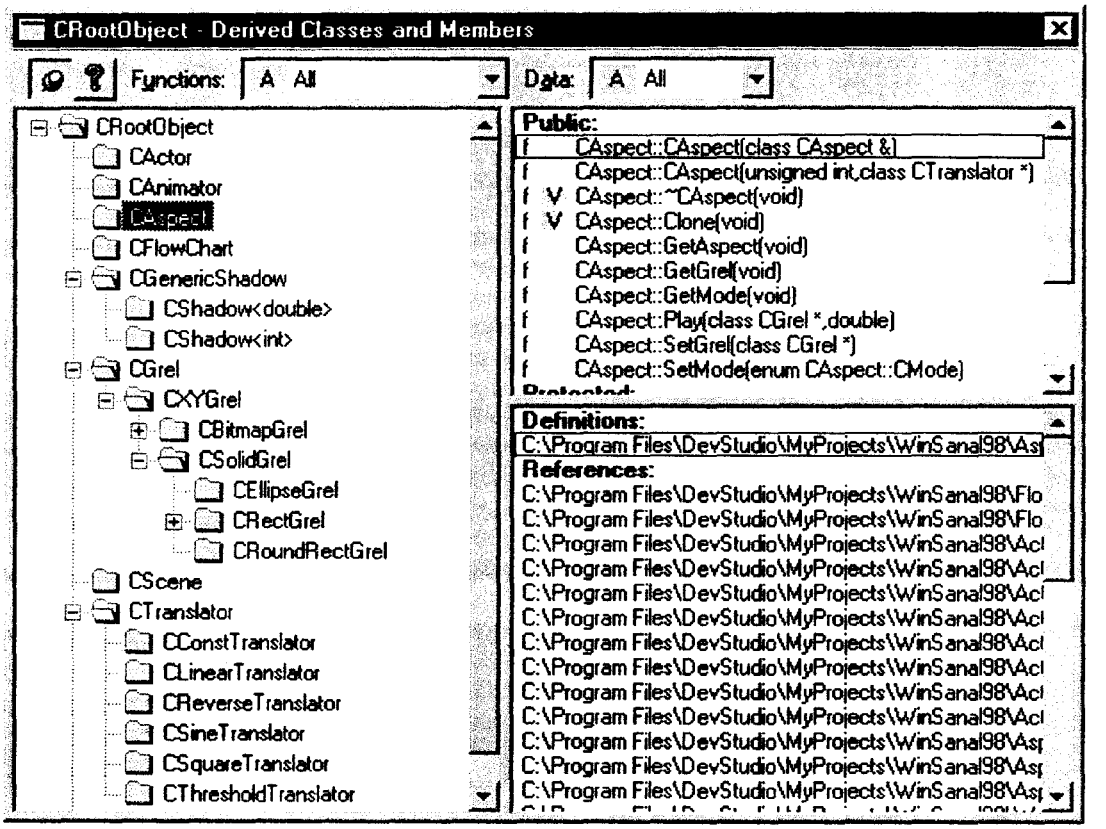
Visual C++ przedstawia zarówno dane statyczne (czasu kompilacji), jak i informacje czasu wykonania. Przeglądarki tekstu źródłowego pozwalają na wyszukiwanie definicji i wszystkich odwołań do wskazanych przez użytkownika symboli oraz grafów funkcji wywołujących i wywoływanych. Wyświetlają także hierarchie klas, dla poszczególnych klas pokazując zmienne i funkcje składowe, wraz z odnośnikami do ich definicji i wystąpień w plikach źródłowych (plansza 3.2). Widoki są zorganizowane w postaci drzew, z możliwością zwijania i rozwijania poszczególnych węzłów, co daje możliwość swobodnego poruszania się po rozbudowanych strukturach i sterowania złożonością obrazu. Informacje czasu wykonania są udostępniane w zakresie zbliżonym do konwencjonalnych debuggerów. Można przeglądać zmienne standardowych typów *C* (*int*, *char*, *float*), zmienne typu *struct* wraz z nazwami pól, a także zmienne wskaźnikowe. Te ostatnie były przez tradycyjne debuggery przedstawiane wyłącznie w postaci heksadecymalnie zapisanych adresów. Narzędzia nowoczesne, do których zalicza się *Visual C++*, wyświetlają dodatkowo niewielki przycisk przy każdym takim adresie; jego kliknięcie pozwala użytkownikowi na szybkie przejście do zmiennej wskazywanej przez dany wskaźnik. W ten sposób można przeglądać nawet bardzo rozbudowane struktury dynamiczne, jak listy, drzewa i sieci.

Visual C++ jest narzędziem dedykowanym do programowania w środowisku *Windows*. Praca programisty w zakresie projektowania aspektów charakterystycznych dla systemów *Windows* oraz *ActiveX* jest ułatwiona dzięki potężnemu narzędziu o nazwie *ClassWizard*. Jest ono równocześnie narzędziem programowania wizualnego, jak i wizualizacji programów, koncentrującym się na różnorodnych relacjach pomiędzy programem a systemem *Windows*, szczególnie tych związanych z programowaniem zdarzeniowym, projektowaniem graficznego interfejsu użytkownika oraz tworzeniem i łączeniem składników (komponentów) w technologii *ActiveX*.

Nawet jeśli możliwości wyświetlania struktur danych bledną w porównaniu z oferowanymi przez *Incense/Pascal Genie*, możliwość szybkiego przeglądania przestrzeni danych i podążania za wskaźnikami daje tego rodzaju wizualizacji wyraźną przewagę nad metodami tradycyjnymi.

3.3 SYSTEMY WIZUALIZACJI ALGORYTMÓW

Obecnie przedstawimy kilka systemów wizualizacji algorytmów, które uznaliśmy za najważniejsze z punktu widzenia naszej pracy. System *BALSA* był nie tylko pierwszym, ale



Plansza 3.2. Przykładowy widok hierarchii klas w zintegrowanym środowisku kompilatora Visual C++ 5.0 firmy Microsoft. Widoczna też lista składowych klasy *CAspect* oraz zestawienie wystąpień konstruktora tej klasy

wciąż jest najlepiej znanym ze wszystkich systemów animacji algorytmów. Pod wieloma względami *BALSA* i jej następcy, *BALSA II* i *ZEUS* są nie do pobicia w konkurencji z innymi rozwiązaniami. System Johna Stasko *Tango* również przyczynił się w ogromnym stopniu do rozwoju dziedziny. System *ANIM* wymienimy ze względu na jego oryginalne podejście do tworzenia wizualizacji algorytmów. Oparty na języku *Smalltalk* system Duisberga *Animus* stanowił wielką inspirację; był on jedną z pierwszych prób stworzenia nieinwazyjnego narzędzia o deklaratywnym stylu specyfikacji wizualizacji. System *UWPI* ma absolutnie wspaniały mechanizm automatycznego generowania abstrakcji. System *Pavane* jest systemem generującym wyjątkowo udane projekcje i reprezentuje najnowsze dokonania w tej dziedzinie. Oczywiście, z konieczności pominięte zostanie wiele systemów, które powinny się tu także znaleźć. Między nimi są: *ALLADIN* [125], *GeoSheet* [126], *PegaSys* [127], *FIELD* [128], *PECAN* [129], *ASAS* [130], oraz systemy pracujące w środowisku Internetu [131, 132].

W tym punkcie opiszemy też systemy *SANAL* i *WinSANAL*, w których stworzeniu braliśmy udział, i które są bezpośrednimi przodkami naszego systemu *Daphnis*. Scharakteryzujemy też system *Siamoa*, zaprojektowany i zaimplementowany przez naszych francuskich współpracowników z Laboratorium Automatyki z Lille. Stanowił on dla nas cenne źródło inspiracji i doświadczeń.

BALSA I ZEUS

Pierwszym poważnym interaktywnym systemem wizualizacji oprogramowania był *Brown University Algorithm Simulator and Animator* (*BALSA*) zaprojektowany przez Marca H. Browna i Roberta Sedgewicka [74, 75]. Ta pionierska praca stała się wzorcem, do którego porównywano wszystkie późniejsze rozwiązania. *BALSA* jest pełnowartościowym produktem, używanym przez setki studentów jako pomoc naukowa w zakresie projektowania i analizy algorytmów [47 – 50].

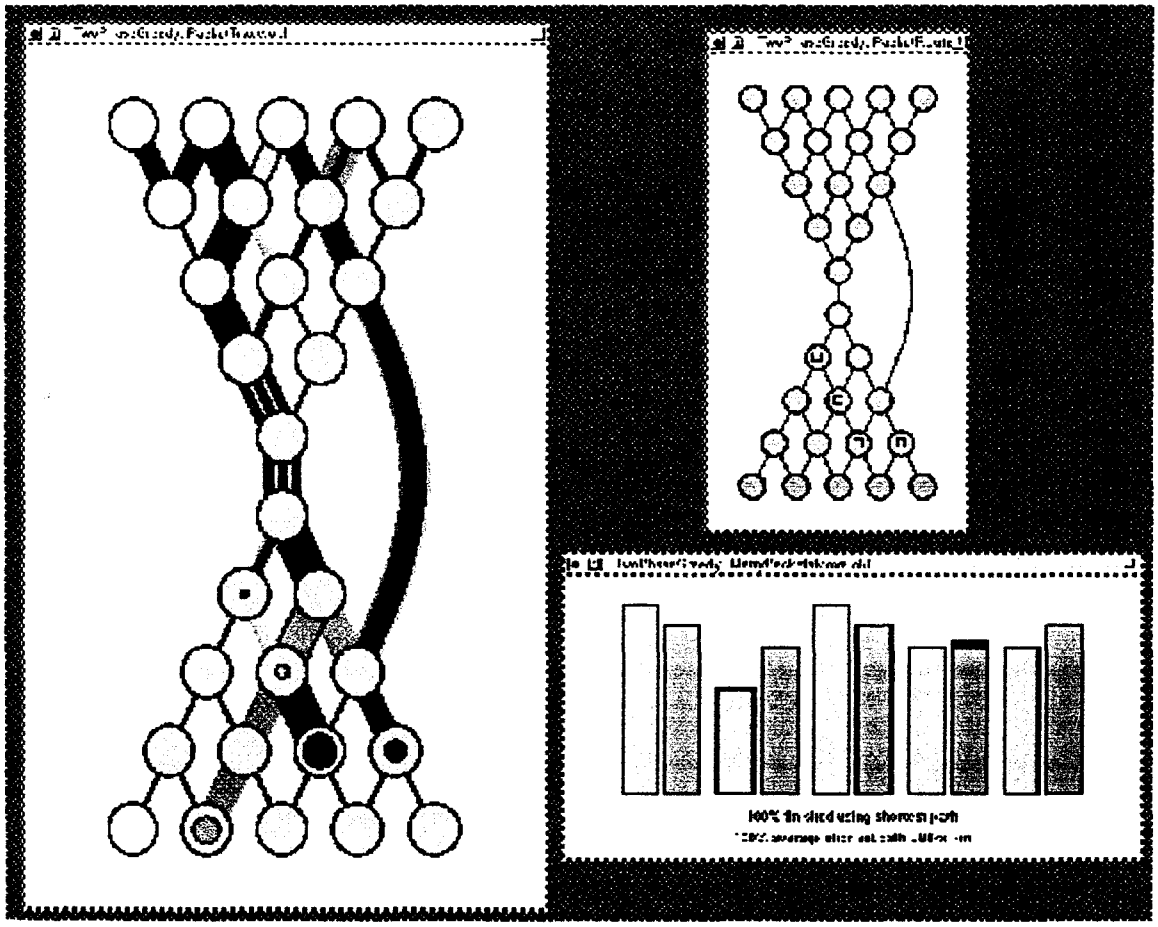
BALSA była jednym z pierwszych systemów korzystających z dużych ekranów graficznych i systemów okienkowych na osobistych stacjach roboczych. Zainstalowana w 1983 roku na 50 stacjach *Apollo* w specjalnie zaprojektowanej dydaktycznej sali laboratoryjnej, umożliwiała studentom obserwację omawianych przez nauczyciela algorytmów na ekranach indywidualnych stacji. Studenci mogli sterować przebiegiem projekcji (inicjować, przerywać, wznawiać, kontrolować tempo, a także dowolnie konfigurować układ okienek), a także zapisywać swoje działania w plikach skryptowych. Dostępny był zestaw gotowych do użycia prezentacji zintegrowany ze znanym podręcznikiem algorytmiki R. Sedgewicka [195].

BALSA została napisana w języku *C*, ale algorytmy, które przedstawia, są na ogół zapisane w *Pascalu*. Zawartość okien wizualizacyjnych jest określana przez wizualizatora i nie może być modyfikowana przez użytkownika – obserwatora. Wizualizacja obejmuje widok sformatowanego tekstu źródłowego, z wyróżnieniem aktualnie wykonywanej linii. Jednak to widoki struktur danych są tym elementem, który zadecydował o powodzeniu systemu. System oferuje szeroki wachlarz dostępnych reprezentacji graficznych danych, od wykresów słupkowych czy punktowych aż po widoki rozbudowanych grafów i drzew i zaawansowane abstrakcje geometryczne. Możliwe też jest przedstawienie różnych widoków tej samej struktury danych, jak też kilku algorytmów wykonywanych jednocześnie. Ta ostatnia funkcja umożliwia tworzenie „wyścigów algorytmów" – podobnych do tych, które swego czasu Baecker zamieścił w swoim *Sorting Out Sorting*.

Wizualizator w systemie *BALSA* wychodzi od tekstu programu w *Pascalu* i w miejscach, gdzie w tekście źródłowym zmianie ulegają interesujące struktury danych, uzupełnia go o wywołania procedur zarządcy zdarzeń interesujących (ang. *interesting events*, por. rozdział 2.5). Następnie tworzy widok odpowiadający konkretnym zdarzeniom. Do dyspozycji jest biblioteka predefiniowanych widoków; możliwe też jest stworzenie własnego widoku. Ostatecznym krokiem jest zdefiniowanie generatora wejściowego, dostarczającego poprawne dane dla programu uruchomionego w środowisku systemu. Użytkownicy mogą uruchamiać animacje i zapisywać swoje działania w skryptach, które następnie można odtwarzać.

Pierwsza wersja systemu *BALSA* pracowała na czarno-białych stacjach *Apollo*. Kolejna wersja, *BALSA-II* [73, 74] przeznaczona była dla kolorowych komputerów Apple Macintosh i pozwalała na ilustrowanie interesujących zdarzeń prostymi efektami dźwiękowymi.

Ostatnim wcieleniem systemu jest Zeus [76 – 78] (plansza 3.3), system przeznaczony do wizualizacji algorytmów zapisanych w Moduli-3 na stacjach roboczych *DEC*. Podobnie jak *BALSA*, również i *Zeus* obsługuje wiele zsynchronizowanych widoków, a także pozwala użytkownikowi na ich edycję. Po zatrzymaniu projekcji można zmienić reprezentację danych. Dostępne są środki do bezpośredniej manipulacji obiektami graficznymi. Po wznowieniu projekcji wszystkie widoki są odświeżane i dalej prezentowane z uwzględnieniem wprowadzonych zmian. *Zeus* oferuje graficzne środki specyfikacji widoków. Dzięki obiektowej implementacji w wielowątkowym, wieloprocesorowym środowisku *Zeus* świetnie się nadaje do wizualizacji algorytmów równoległych. Na uwagę zasługuje wyposażenie systemu w możliwości generowania dźwięku przy wykorzystaniu syntezatora *MIDI*; prace Browna prowadzo-



Plansza 3.3. Przykładowa wizualizacja przygotowana przy pomocy systemu ZEUS: symulacja zestawienia połączeń sieciowych [77]

ne wraz z Hershbergerem [19] są świetnym przyczynkiem do wykorzystania koloru i dźwięku w animacji algorytmów. Jednym z najnowszych rozszerzeń i udoskonaleń systemu jest wprowadzenie możliwości tworzenia prezentacji trójwymiarowych [18].

Mimo że *BALSA* jest właściwie prekursorem w dziedzinie animacji algorytmów, jakość tworzonych przy jej pomocy projekcji do dzisiaj wyznacza bardzo wysoki standard, nieosiągalny dla wielu późniejszych rozwiązań. System *Zeus*, ze swoją wyrafinowaną, trójwymiarową grafiką, ustawił poprzeczkę jeszcze wyżej. Środki dostępne w obydwu systemach pozwalają na tworzenie wizualizacji na bardzo wysokim poziomie abstrakcji. Z drugiej strony przygotowanie projekcji jest dość uciążliwe – wymaga bowiem żmudnego wprowadzania do tekstu źródłowego wywołań procedur zgłaszających zdarzenia interesujące.

TANGO, XTANGO I POLKA

System *Tango* [80, 82], i jego następca, *XTango* [83, 85], podobnie jak *BALSA* stworzone w *Brown University*, zostały zaprojektowane i zbudowane przez Johna Stasko. Dzięki zastosowaniu oryginalnego „paradygmatu" ścieżka – tranzycja (ang. *path – transition paradigm*) [81], udało się w systemie *Tango* uzyskać płynną animację wysokiej jakości, bez potrzeby oprogramowywania kolejnych kadrów. Paradygmat ten zastąpił prostą, ale żmudną dla wizualizatora technikę opartą na zmazywaniu i rysowaniu, stosowaną we wcześniejszych rozwiązaniach.

Tworzenie animacji składa się z trzech faz: określenia zestawu abstrakcyjnych operacji i zdarzeń w programie, które stanowić będą element sterujący projekcją, zaprojektowania efektów graficznych i animacyjnych je symulujących i wreszcie stworzenia odwzorowania operacji abstrakcyjnych na reprezentacje graficzne.

Pierwsza ze wspomnianych faz obejmuje uzupełnienie tekstu źródłowego programu o wywołania tak zwanych *operacji algorytmu*. Może to być wykonane ręcznie albo za pomocą specjalnego edytora.

Druga faza – projektowanie animacji – związana jest z wykorzystaniem paradygmatu ścieżka – tranzycja. W jego ramach wprowadzono cztery abstrakcyjne typy danych, pozwalające na wyspecyfikowanie efektów graficznych i animacyjnych. Dostępne jest także prostsze w obsłudze narzędzie graficzne, które automatycznie generuje niezbędną specyfikację wizualizacji na podstawie zademonstrowanego przez użytkownika przykładu animacji [79].

W trzeciej fazie następuje przyporządkowanie animacji odpowiednim operacjom algorytmu. Możliwe jest przyporządkowanie tego samego efektu animacyjnego różnym operacjom algorytmu (jest to nowa możliwość w porównaniu z systemem *BALSA*).

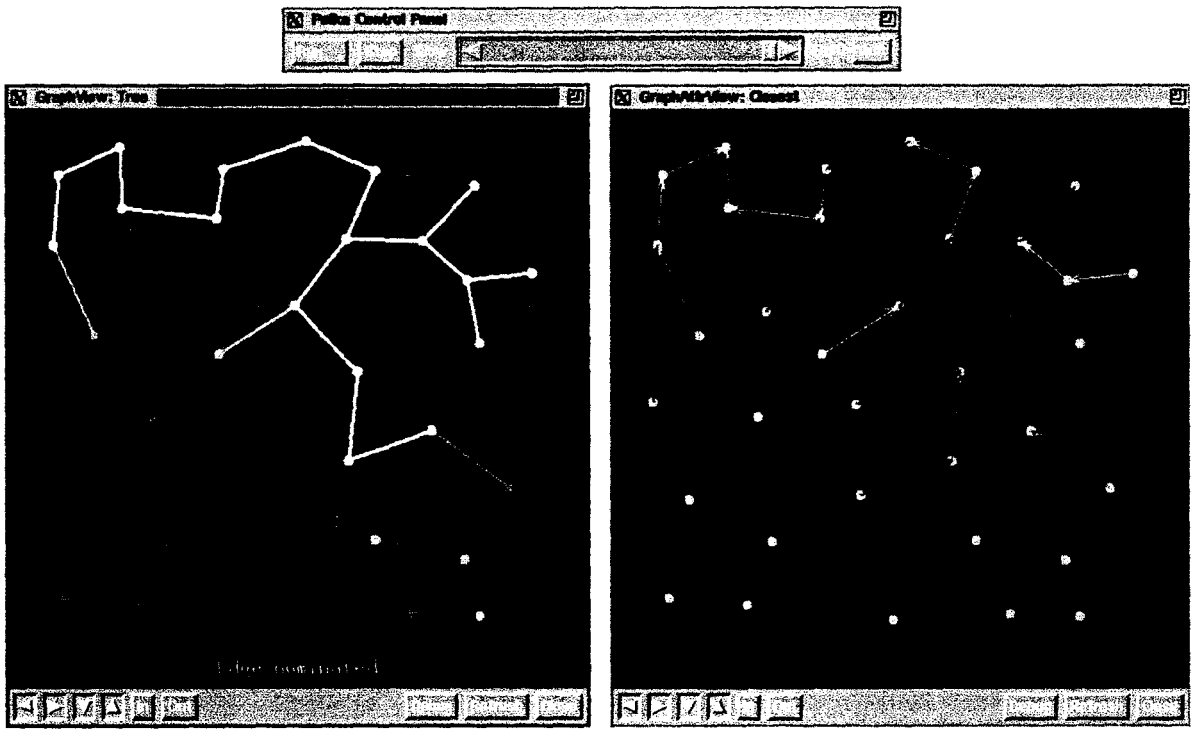
Aby dokonać animacji programu za pomocą systemu *Tango*, wizualizator najpierw uzupełnia tekst źródłowy wizualizowanego programu wywołaniami operacji algorytmu. Następnie tworzone są *sceny animacji*, które w ostatecznej formie specyfikacji są funkcjami języka *C*, przyjmującymi argumenty pochodzące z programu poddawanego wizualizacji. Ostatnim krokiem jest stworzenie pliku sterującego animacją, który definiuje odwzorowanie operacji algorytmu na sceny animacji. Po wykonaniu tego kroku animacja jest gotowa do użycia. Użytkownik ma do dyspozycji elementarne narzędzia do sterowania okienkami, a także zatrzymywania i wznawiania projekcji oraz regulacji jej tempa.

Oryginalna wersja systemu była zaimplementowana na stacjach *BSD Unix* i korzystała ze specjalnego pakietu graficznego, zaprojektowanego w *Brown University* i przeznaczonego do tworzenia grafiki dwuwymiarowej. Nowsza wersja, *XTango*, daje się uruchamiać na większości stacji unixowych obsługujących system okienkowy X11. Jest ona udostępniona szerokiemu kręgowi odbiorców, wraz z obszerną biblioteką gotowych animacji.

System *POLKA* [84, 85] (plansza 3.4) to następca *XTango*; jest znacznie bardziej elastyczny i rozbudowany. *POLKA* jest systemem animacji ogólnego przeznaczenia, szczególnie dobrze dostosowanym do animacji programów i algorytmów. Obecnie wszystkie prace wykonywane w zespole Johna Stasko są już prowadzone na bazie systemu *POLKA*, nie *XTango*. System był budowany z myślą o wizualizacji programów i algorytmów równoległych, nadaje się jednak też i do szeregowych. *POLKA* dostarcza szeregu abstrakcji wysokiego poziomu, które ułatwiają i przyspieszają tworzenie animacji.

POLKA obejmuje interaktywny moduł o nazwie *SAMBA*. Jest to interpretator animacji, odczytujący i wykonujący komendy podane w postaci tekstowej. Komendy te są raczej elementarne, i autorzy rozwiązania twierdzą, że są bardzo łatwe w użyciu. Program napisany w dowolnym języku można łatwo wzbogacić o instrukcje generujące komendy tekstowe, które skutecznie sterują maszyną projekcyjną modułu *SAMBA*. W ten sposób można przygotować wizualizację bez względu na język, w którym zapisano algorytm – i jest to jedna z głównych zalet tego modułu. Rozwiązanie to przypomina system *ANIM*, który opisujemy poniżej.

Podsumowując, systemy *Tango/Xtango* oraz ich następca, *POLKA*, umożliwiają tworzenie projekcji charakteryzujących się bardzo wysokim poziomem abstrakcji. Przygotowanie projekcji wydaje się nieco mniej uciążliwe, niż w przypadku systemów *BALSA/Zeus*, w dalszym ciągu wymaga jednak sporo wysiłku i ingerencji w tekst źródłowy programu.



Plansza 3.4. Przykładowa wizualizacja przygotowana przy pomocy systemu POLKA: równoległy algorytm wyszukiwania drzewa rozpinającego [85]

ANIM

ANIM [86, 87] jest prostym ale potężnym systemem wizualizacji oprogramowania, przeznaczonym do tworzenia wizualizacji zarówno dynamicznych, jak i statycznych (tzw. *snapshot*). Zaprojektowany w *AT&T Bell Laboratories ANIM* jest zgodny z filozofią systemu *UNIX*: posiada silny a zarazem prosty interfejs, całkowicie niezależny językowo i sprzętowo.

ANIM jest systemem typu *post-mortem*, pozwalającym na tworzenie projekcji lub pojedynczych obrazów na podstawie skryptu, który jest generowany przez wizualizowany program podczas wykonania. Skrypt ten buduje się z ośmiu prostych komend tekstowych (cztery komendy graficzne: *line*, *text*, *box* i *circle* oraz cztery komendy sterujące: *view*, *click*, *erase* i *clear*). Ponieważ skrypt jest zwykłym plikiem tekstowym, może być łatwo wygenerowany przez program napisany w dowolnym języku – uzupełnienie tekstu źródłowego sprowadza się więc do wprowadzenia instrukcji zapisu ciągów znakowych do pliku skryptu. Decyduje to o niezależności językowej systemu (por. z systemem *POLKA*). Przetłumaczenie i wykonanie uzupełnionego programu powoduje wytworzenie skryptu, który po zakończeniu programu (a więc w trybie *post-mortem*) może być użyty do sterowania wizualizacją. Sam system stanowi więc rodzaj interpretatora plików skryptowych.

Technika *post-mortem* została wybrana ze względu na możliwość łatwej obróbki skryptów przed uruchomieniem projekcji, w szczególności za pomocą standardowych narzędzi *Unixa*, na przykład *awk*. Po zakończeniu wykonania programu znany jest obszar roboczy niezbędny do przedstawienia projekcji, co umożliwia właściwe wyskalowanie obrazu. Przetworzenie kilku skryptów za pomocą niewielkiej liczby linijek w *awk* pozwala na synchroniczną prezentację kilku widoków, na przykład w celu sporządzenia wyścigu algorytmów.

W podanym przez autorów systemu przykładowym zastosowaniu systemu stworzono animowaną ilustrację dla krótkiego wykładu o sortowaniu (dla którego klasyczny film *Sorting Out Sorting* okazał się za długi i za dokładny). Efektem pracy było pięciominutowe nagranie wideo. Zabrało to dwie godziny na stworzenie skryptu za pomocą 74 linii *awk* i następne dwie godziny na techniczne przygotowanie taśmy – razem cztery godziny. Dla porównania, *Sorting Out Sorting*, wprawdzie zdecydowanie bardziej rozbudowany i sześciokrotnie dłuższy, zabrał jednak swoim twórcom aż cztery lata pracy.

System *ANIM* był jeszcze wielokrotnie skutecznie wykorzystywany, między innymi do trójwymiarowego modelowania układów molekularnych, tworzenia obrazów stereoskopo-

wych, wizualizacji analiz numerycznych, równań różniczkowych, uruchamiania programów macierzowych, statystycznych, a także do wizualizacji programów równoległych.

Podsumowując, *ANIM* oferuje prosty sposób generowania animacji, dobrze wkomponowany w system *UNIX*. Jego ogólny interfejs wymaga co prawda bardzo szczegółowego, czysto imperatywnego specyfikowania wizualizacji, wydaje się jednak skuteczny zarówno w zastosowaniach dydaktycznych, jak i badawczych. Wizualizacje o wysokim stopniu abstrakcji są wprawdzie możliwe, wymagają jednak znacznego nakładu pracy.

ANIMUS

Wzorując się na systemie *BALSA*, Ralph London i Robert A. Duisberg zastosowali w 1985 imperatywną metodę specyfikacji – uzupełnienie algorytmów o wywołania procedur zgłaszających interesujące zdarzenia – w celu zrealizowania wizualizacji kilku algorytmów zapisanych w języku Smalltalk za pomocą systemu o nazwie *Animus* [88 – 91].

Późniejszym, oryginalnym dokonaniem Duisberga było wykorzystanie ograniczeń czasowych (ang. *temporal constraints*) do opisu wyglądu i struktury obrazu oraz zmian, którym obraz ten ulega w czasie. Animacja algorytmów była jednym z zastosowań tej techniki. Ograniczenia były specyfikowane poza programem; wyzwalane były one przez przesyły komunikatów *Smalltalka*. Umożliwiło to animację algorytmów bez potrzeby modyfikacji tekstu źródłowego, poprzez osobną definicję ograniczeń wyzwalających i sterujących projekcją [89]. Tym samym system Duisberga był pierwszym, w którym wykorzystano deklaratywną metodę specyfikacji wizualizacji, i to na dodatek całkowicie nieinwazyjną.

Część autorów wskazuje jednak na pewne istotne słabości tego podejścia [74]. Otóż Duisberg pokazuje działanie systemu na przykładzie algorytmów sortowania. System ogranicza się w zasadzie do monitorowania komunikatów *compareWith:* i *exchangeWith:*, które zostały użyte do implementacji algorytmów. W ten sposób wizualizacja może być skutecznie przeprowadzona bez zmiany tekstu źródłowego. Jednak, jak zauważa Marc H. Brown w swojej pracy doktorskiej, „jest to możliwe tylko dzięki temu, że wyzwalacze sterujące animacją akurat pokrywają się z komunikatami użytymi w implementacji algorytmu, i komunikaty te akurat okazały się właściwymi abstrakcjami. Jeśliby algorytm był zaimplementowany z użyciem komunikatów *setValue:* zamiast *exchangeWith:* taka animacja nie byłaby możliwa bez zmiany tekstu źródłowego" [74]. Mimo tych wątpliwości, uznać musimy pionierską rolę systemu *Animus* jako pierwszego przyczynku w kierunku deklaratywnego stylu specyfikacji wizualizacji.

UWPI

System o nazwie *University of Washington Program Illustrator* (*UWPI*) [92], zaimplementowany w języku *Lisp*, automatycznie wizualizuje abstrakcyjne struktury danych wysokiego poziomu, zaprojektowane przez użytkownika. Nadaje się on do animowania programów napisanych w języku będącym podzbiorem *Pascala*, obejmującym instrukcje blokowe i sterujące, deklaracje stałych oraz zmienne całkowite, logiczne, wskaźnikowe, rekordowe oraz tablicowe jedno- i dwuwymiarowe. Użytkownicy wprowadzają tekst źródłowy do systemu, który dokonuje analizy i następnie rozpoczyna (automatycznie) projekcję. W jej trakcie wyświetlany jest sformatowany listing programu z zaznaczeniem aktualnie wykonywanej linii. Okno widoku danych przedstawia abstrakcyjny widok struktur danych, będący wynikiem poprzedniej automatycznej analizy (plansza 3.5).

Sercem systemu *UWPI* jest moduł wnioskujący (ang. *inferencer*), który analizuje każdą napotkaną w tekście źródłowym strukturę danych i proponuje kilka odpowiednich abstrakcji danych. Każdej z nich przydzielana jest waga, oparta na zgodności pomiędzy repertuarem operacji dostępnych dla tej abstrakcji, a operacjami faktycznie odnalezionymi w tekście źródłowym. Następnie, dla każdej struktury danych, abstrakcja o najwyższej końcowej wadze jest przekazywana do modułu strategii rozkładu planarnego (ang. *layout strategist*). Generuje on właściwe dla poszczególnych abstrakcji reprezentacje graficzne i umieszcza je na ekranie.

W zrealizowanej wersji prototypowej systemu zaimplementowano abstrakcyjne reprezentacje dla wartości logicznych i skalarnych, wskaźników, indeksów, relacji (digrafów), kolejek i list. Pozwalało to na dość efektywną wizualizację algorytmów sortowania i przeszukiwania grafów. Z drugiej strony, *UWPI* nie był przeznaczony do pracy z dużymi zestawami danych, nie pozwalał na tworzenie nowych abstrakcji danych, a używane reprezentacje graficzne były zapożyczone z innych systemów wizualizacji.

Poziom abstrakcji projekcji generowanych przez *UWPI* nie dorównuje może osiąganemu przez takie systemy, jak *BALSA* czy *XTango*, ale automatyczny tryb ich tworzenia jest naprawdę imponujący. System *UWPI* wydaje się posiadać wysoki poziom wiedzy o programie, pozwalający na tworzenie abstrakcyjnych reprezentacji graficznych struktur danych bez pomocy ze strony użytkownika. Metoda akwizycji wiedzy zastosowana w systemie jest jednak przez samych autorów określana jako nieformalna; rozpoczynali oni od pewnych przybliżonych reguł i dostrajali je tak, by były zadowalające w konkretnych przykładach. System dopasowuje raczej wyrywkowe dane o programie do zestawu reguł. Niemniej jednak *UWPI* jest

prawdopodobnie najciekawszym systemem wizualizacji stosującym tak daleko idące metody analizy programów.

UWPI pozostał na poziomie wersji prototypowej. Chociaż zastosowane w nim rozwiązania wydają się bardzo obiecujące, nie podjęto jednak dalszych prac rozwojowych nad systemem, głównie z przyczyn finansowych.

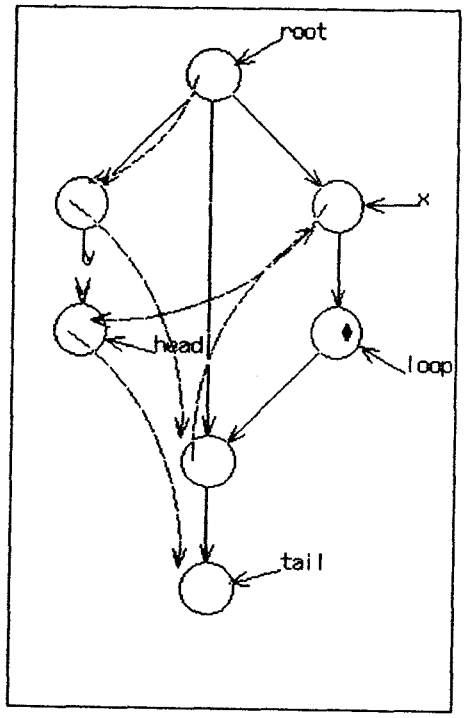
PAVANE

System o nazwie *Pavane* został stworzony przez zespół z Washington University w St. Louis [93 – 99] (plansza 3.6). Przeznaczony jest on do tworzenia trójwymiarowych wizualizacji programów równoległych napisanych w języku *Swarm* [96], z późniejszymi rozszerzeniami pozwalającymi na wizualizację programów napisanych w języku *C*. Charakterystyczną cechą systemu *Pavane* jest w pełni deklaratywny styl specyfikacji wizualizacji.

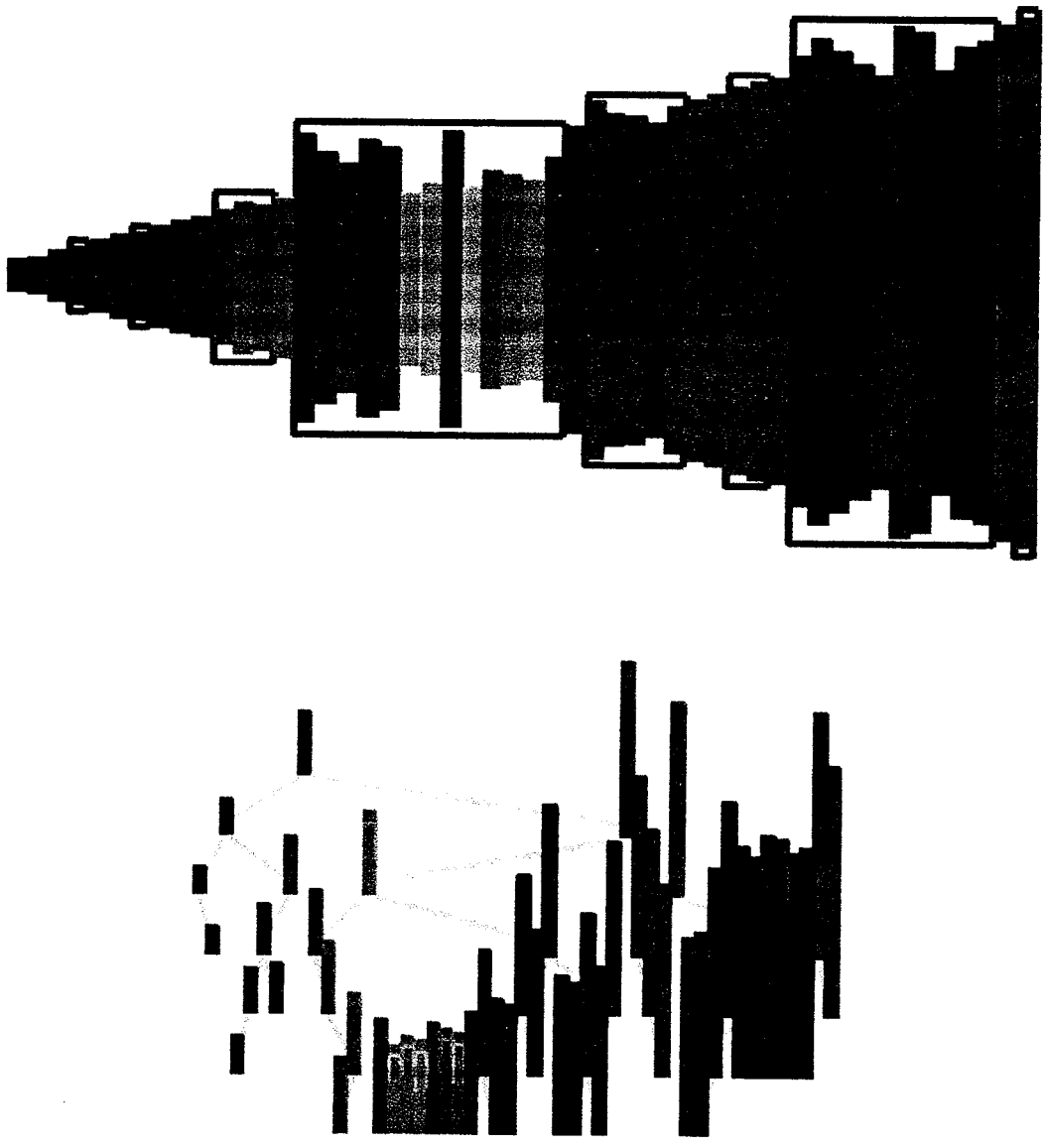
System składa się z pięciu części: parsera tłumaczącego programy w języku *Swarm* i specyfikacje wizualizacji na język pośredni (pierwotnie *Prolog*, później *C*), biblioteki czasu wykonania niezbędnej do wykonywania programów w języku *Swarm*, biblioteki wykorzystywanej przy wizualizacji programów w *C*, dodatkowej biblioteki czasu wykonania niezbędnej do wykonywania skompilowanych wizualizacji zapisanych w języku pośrednim, które tworzą ślad wykonania, oraz programu uwidaczniającego wizualizacje i dostarczającego interfejs użytkownika. Projekcje realizowane przez ten ostatni program są sterowane wytworzonym uprzednio śladem wykonania, tak więc *Pavane* jest systemem typu *post-mortem*.

Wszystkie komponenty systemu są zaimplementowane w języku *C* pod kontrolą systemu *Unix*. Proces wytworzenia wizualizacji składa się z trzech faz: utworzenia opisu wizualizacji w języku pośrednim, obliczenia wizualizacji (wytworzenia śladu wykonania) i wyświetlenia wizualizacji.

Wizualizacje są w systemie *Pavane* specyfikowane w stylu deklaratywnym [97], w którym wizualizator definiuje transformacje pomiędzy stanem procesu obliczeniowego a końcowym obrazem graficznym. W najprostszym przypadku transformacja taka składa się z deklaracji obiektów graficznych, których parametry mogą być zmieniane przez operacje programu. Jednak *Pavane* dostarcza też mocniejszych możliwości tworzenia abstrakcji. Transformacja jest wyrażona serią odwzorowań między przestrzeniami wejściową i wyjściową, przy czym każde odwzorowanie jest zdefiniowane co najmniej jedną regułą. Reguła określa relację pomiędzy dwiema przestrzeniami – na przykład, *dla każdej trójki (i, j, v) w przestrzeni wejścio-*



Plansza 3.5. Widok struktury danych wygenerowany przez system UWPI [11]



Plansza 3.6. Dwa widoki algorytmu szybkiego sortowania, uzyskane za pomocą systemu *Pavane* [98]

wej istnieje jednostkowy prostopadłościan (róg = (i, j, 0), szerokość = 1, wysokość = 1, głębokość = v) w przestrzeni wyjściowej.

System *Pavane* posiada wiele udogodnień przeznaczonych specjalnie do wizualizacji programów równoległych. Jednakże ze względu na fakt, że wizualizacja takich programów wiąże się z wysoce specyficzną problematyką, wykraczającą poza ramy tej pracy, nie będziemy przedstawiać tych elementów systemu.

System *Pavane* jest nie tylko jednym z najnowszych, ale i najlepszych rozwiązań w zakresie animacji algorytmów. Możliwości tworzenia wysoce abstrakcyjnych wizualizacji są imponujące i stawiają ten system w ścisłej czołówce systemów. Przygotowanie projekcji za pomocą specyfikacji deklaratywnej nie wymaga modyfikacji tekstu źródłowego programu i jest stosunkowo łatwe.

Whizz i Witness

System *Whizz* [100, 101] opracowany przez Stéphana Chatty z Centre d'Orsay Uniwersytetu Paris-Sud w 1992 roku jest narzędziem do budowy animowanych interfejsów użytkownika. Stanowi on rozszerzenie systemu X_{TV}, służącego do tworzenia interfejsów użytkownika korzystających z techniki bezpośredniej manipulacji elementami graficznymi [102]. Najważniejszym zastosowaniem systemu jest *Witness* – wizualny debugger i system animacji algorytmów.

System *Whizz* oparty jest na zasadzie obiektów komunikujących się za pomocą strumieni danych i zdarzeń. Pewne obiekty inicjują animację poprzez akt spontanicznej emisji informacji. Model ten umożliwia jednolity opis dynamicznego zachowania się określonego interfejsu użytkownika, bez względu na to, czy kolejne obrazy są tworzone wyłącznie w funkcji czasu, czy też są sterowane działaniami użytkownika, czy wreszcie są ilustracją (wizualizacją) zmieniających się danych konkretnej aplikacji. Właśnie ten trzeci przypadek znalazł zastosowanie w systemie *Witness*. Obiekty inicjujące animację to zmienne poddawanego wizualizacji programu, a ściślej – odpowiadające im twory po stronie systemu. W oryginalnej zastosowanej przez Chatty metaforze muzycznej zmienne te to „instrumenty muzyczne", a generowany przez nie strumień danych definiujących zmiany wartości tych zmiennych to „nutki". Nutki wpływają bezpośrednio na sposób zachowania się „tancerzy", którzy odpowiadają elementom graficznym na ekranie. Pojęcia „tempa" i „rytmu" odnoszą się do temporalnych ograniczeń prezentowanego obrazu.

Witness, będąc w zamierzeniu wizualnym debuggerem, pozwala na przygotowanie wizualizacji w sposób stosunkowo prosty, choć pod względem poziomu abstrakcji uzyskiwanych projekcji ustępuje on znacznie czołowym systemom animacji algorytmów.

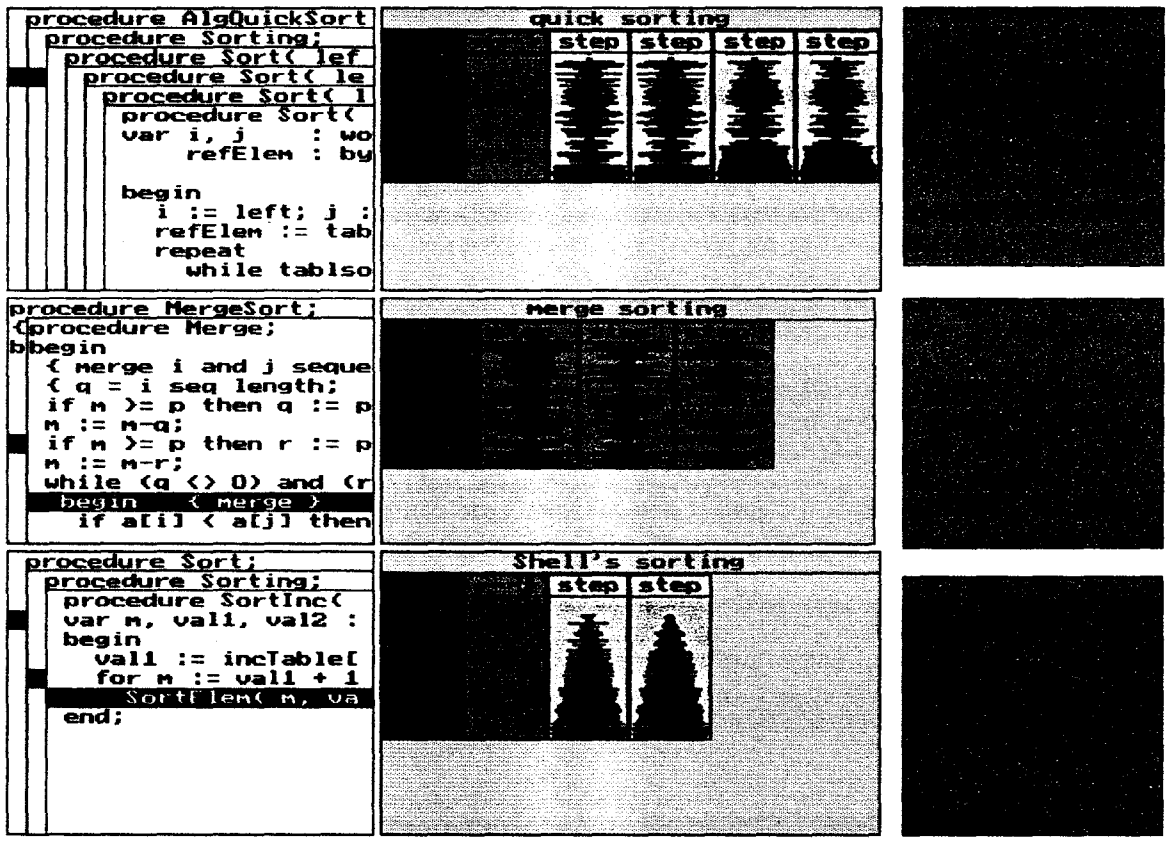
Na podkreślenie zasługuje fakt, że *Whizz* to system, którego przeznaczenie znacznie wykracza poza wizualizację algorytmów. *Witness* – narzędzie do animacji algorytmów – jest tylko jednym z wielu możliwych zastosowań. Wizualizacja oprogramowania staje się w ten sposób tylko przypadkiem szczególnym szerszej klasy problemów, jaką stanowią animowane interfejsy użytkownika. Do innych niż wizualizacja skutecznych zastosowań systemu *Whizz* należy interfejs systemu kontroli ruchu lotniczego, którego istotnym elementem jest podsystem bezpośredniej manipulacji danymi. Obiektami inicjującymi animację – instrumentami muzycznymi – są w tym przypadku z jednej strony dane przekazywane z radaru, z drugiej zaś – decyzje podejmowane przez kontrolera.

SANAL

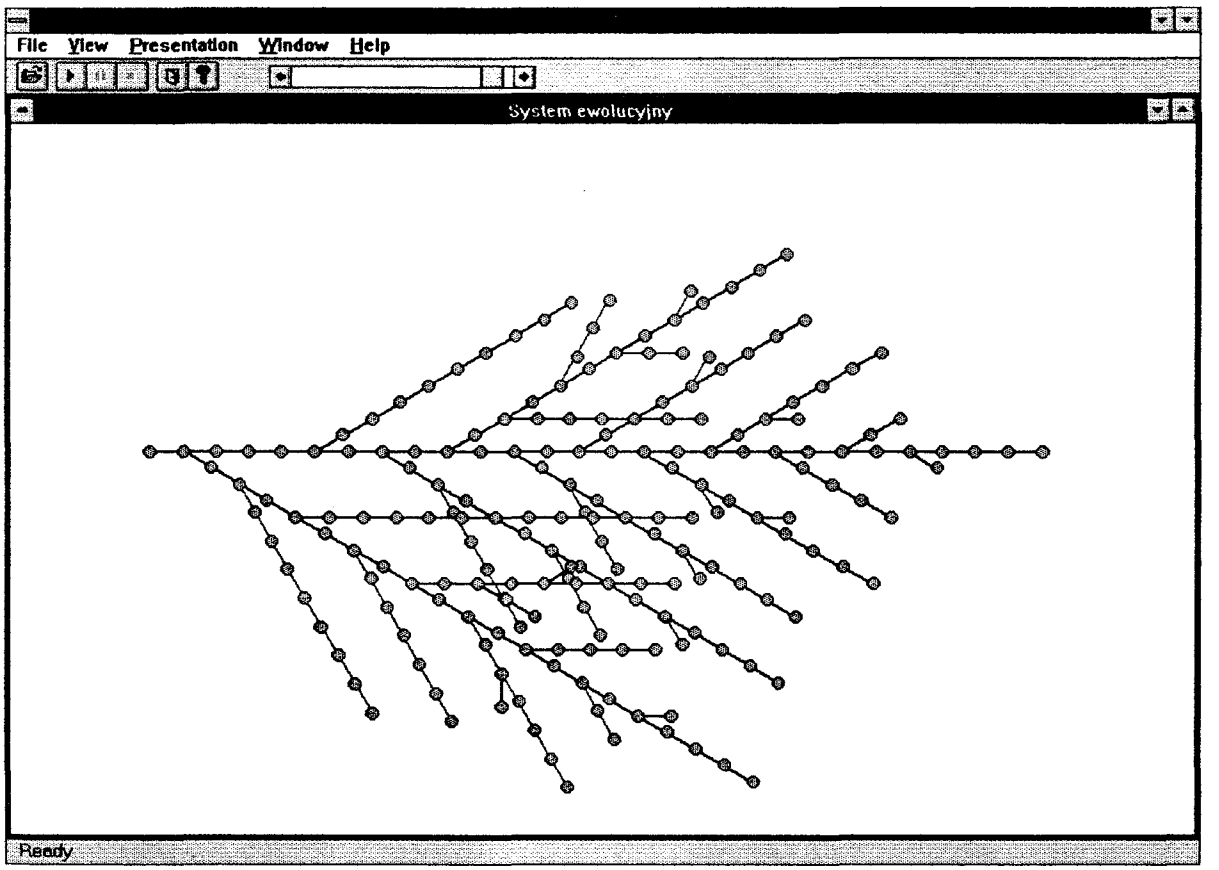
System animacji algorytmów *SANAL* (plansza 3.7) powstał w Instytucie Informatyki Politechniki Śląskiej w roku 1989 w zespole pod kierownictwem Przemysława Szmala [104, 105]. Kolejne wersje tworzono w latach 1990-1993. Nasze prace związane z animacją algorytmów rozpoczęły się w 1993 roku właśnie od utworzenia wersji *3.0* systemu *SANAL*. Jest to wersja ostateczna systemu. Podobnie jak wersje poprzednie, napisana jest w języku *Turbo Pascal*. System jest przystosowany do pracy na komputerach zgodnych z *IBM PC* pod kontrolą systemu operacyjnego *MS-DOS*.

System jest wyposażony w obszerny zestaw środków prezentacji graficznej danych i przebiegu sterowania programów. Posiada mechanizmy pozwalające na dostosowanie treści i tempa projekcji do potrzeb użytkownika oraz narzędzia ułatwiające przygotowanie programów do pokazu — dostosowanie ich do współpracy z systemem. W skład systemu SANAL wchodzi też zestaw programów gotowych do projekcji.

Przygotowanie algorytmów do animacji w systemie SANAL przebiega w kilku etapach. Algorytm musi być dostarczony w formie programu napisanego w *Turbo Pascalu*. Po zdecydowaniu, co i w jaki sposób powinno zostać zobrazowane podczas projekcji, do tekstu programu są wprowadzane wywołania funkcji usługowych systemu SANAL. Dalsza obróbka jest wykonywana automatycznie i obejmuje: wstawienie dodatkowych instrukcji służących do wizualizacji przepływu sterowania programu, kompilację i uruchomienie otrzymanego programu.



Plansza 3.7. Wyścig algorytmów, przygotowany przy pomocy systemu SANAL: porównanie trzech zaawansowanych algorytmów sortowania: szybkiego, przez łączenie i Shell'a



Plansza 3.8. Przykładowa wizualizacja przygotowana przy pomocy systemu WinSANAL: symulacja zachowania systemu ewolucyjnego.

Wstawianie wywołań funkcji usługowych do tekstu samodzielnie animowanego programu wymaga od użytkownika wiedzy na temat ogólnych zasad zarządzania projekcją i mechanizmu wizualizacji, a także zawartości biblioteki funkcji usługowych systemu. Uzyskiwane animacje charakteryzują się dość wysokim poziomem abstrakcji, choć często wymagają dość skomplikowanych modyfikacji i uzupełnień tekstów źródłowych programów poddawanych wizualizacji.

W aktualnej wersji systemu SANAL wprowadzono środki umożliwiające synchroniczną prezentację kilku algorytmów w tym samym czasie. Każdy z tych algorytmów jest zaimplementowany w postaci sekwencji instrukcji, będących wydzieloną procedurą lub programem, które są wykonywane jednocześnie, przynajmniej w subiektywnym odczuciu użytkownika. Umożliwia to przeprowadzenie wyścigów algorytmów, takich jak przedstawione na planszy 3.7 porównanie zaawansowanych algorytmów sortowania. Program przygotowany do prezentacji indywidualnej nadaje się w systemie SANAL do projekcji synchronicznej bez żadnych modyfikacji, i może być ewentualnie wzbogacony o dodatkowe wywołania funkcji związanych z precyzyjną synchronizacją procesów. Przygotowanie projekcji synchronicznej wymaga od użytkownika jedynie wskazania (w trybie interakcyjnym) tego, prezentację których algorytmów należy zrealizować, i, ewentualnie, w jakim trybie, tempie *etc.*

System SANAL stanowi od 1989 swojego rodzaju poligon doświadczalny dla prac dotyczących wizualizacji algorytmów prowadzonych w Instytucie Informatyki Politechniki Śląskiej. Między innymi posłużył do badań nad rozproszoną wizualizacją wielostanowiskową [103].

WinSanal

System *WinSANAL* (plansza 3.8), bezpośredni poprzednik systemu *Daphnis*, został stworzony na bazie doświadczeń zdobytych w trakcie realizacji i rozwoju systemu *SANAL* [107 – 110, 114]. Celem prac była realizacja systemu o architekturze w pełni obiektowej, ze sterowaną danymi metodą wizualizacji i o deklaratywnej metodzie specyfikacji, w przeciwieństwie do systemu *SANAL*, opartego na koncepcji interesujących zdarzeń.

WinSANAL pracuje w środowisku *Windows 95/NT 4.0*. Został zaimplementowany w języku C++ i jest przeznaczony do wizualizowania programów zapisanych w tym języku. W jednej z wersji systemu użytkownik może interaktywnie modyfikować zawartość obserwowanych struktur danych albo przez bezpośrednią manipulację elementami graficznymi na ekranie, albo też symbolicznie.

Algorytm przeznaczony do prezentacji musi być dostarczony w formie programu napisanego w języku C lub C++. Przygotowanie do projekcji przebiega w dwóch etapach. W pierwszym, po zdecydowaniu, co i w jaki sposób powinno zostać zobrazowane podczas projekcji, do tekstu programu wprowadzane są wywołania funkcji usługowych systemu *WinSANAL*. Tak zmodyfikowany program poddawany jest kompilacji. W drugim etapie przygotowuje się tzw. skrypt konfiguracyjny, który informuje system o transformacjach, jakim mają być poddane wartości poszczególnych zmiennych, oraz o reprezentacji graficznej, jaka ma być zastosowana do ich pokazania. Do zmodyfikowanego i skompilowanego algorytmu można zastosować różne schematy wizualizacji, poprzez prostą wymianę skryptu konfiguracyjnego.

W obecnej wersji systemu cały proces przygotowania programu do projekcji jest wykonywany ręcznie, aczkolwiek znaczną część operacji można przeprowadzić mechanicznie z wykorzystaniem istniejących wzorcowych plików konfiguracyjnych.

System jest wyposażony w obszerny zestaw środków prezentacji graficznej i tłumaczenia danych. Do prezentacji indywidualnych elementów danych wykorzystywane są grele – standaryzowane elementy graficzne. Dostępne są między innymi grele w kształcie prostych figur geometrycznych (np. prostokąty, elipsy), bitmapowe (tj. wyświetlane w postaci mapy bitowej), a także obiekty przeznaczone do reprezentowania zmiennych znakowych, tekstowych, oraz wierzchołków i krawędzi struktur grafowych. Obrazy złożonych struktur danych konstruuje się z wykorzystaniem połączonych zespołów greli elementarnych.

Tłumaczenie wartości danych na odpowiadające im atrybuty greli realizują wewnętrzne translatory. Dostępne obecnie kategorie tych obiektów realizują proste przekształcenia funkcyjne (np. liniowe, sinusoidalne), progowanie, porównywanie wartości itp. Translatory mogą być też stosowane do kształtowania torów, po których poruszają się obiekty ruchome (domyślnie przyjmuje się tory prostoliniowe).

Opisane mechanizmy sprzyjają tworzeniu projekcji charakteryzujących się bardzo wysokim poziomem abstrakcji. Projekcje, które w większości systemów byłyby możliwe tylko pod warunkiem wprowadzenia znacznych uzupełnień do tekstu źródłowego programu, w systemie *WinSANAL* uzyskuje się poprzez odpowiednie skonfigurowanie obiektów tłumaczących wartości danych na wartości atrybutów greli. Sprzyja to stworzeniu mechanizmów deklaratywnej specyfikacji wizualizacji. Z tej możliwości skorzystaliśmy jednak w pełni dopiero w systemie *Daphnis* – następcy *WinSANALa*.

Podczas prezentacji użytkownik może aktywnie ingerować w wykonanie obserwowane

programu. Może on modyfikować wartości zmiennych prezentowanych na ekranie, albo oddziałując z obrazami reprezentującymi wartości, albo wpisując nowe dane w tradycyjnej, symbolicznej postaci korzystając z odpowiedniego okienka dialogowego. Jak wspominaliśmy, możliwości te nie są dostępne we wszystkich wersjach systemu. Natomiast w standardowym trybie interakcyjnym można dostosowywać do swoich potrzeb tempo i inne parametry pokazu.

Wraz z systemem WinSanal dostarczana jest biblioteka przygotowanych do animacji programów, realizujących klasyczne algorytmy z takich dziedzin, jak sortowanie, wyszukiwanie, obliczenia numeryczne, systemy ewolucyjne, automatyczna translacja i inne.

Doświadczenia wyniesione z prac nad systemami *SANAL* oraz *WinSANAL* zostały wykorzystane też w innym projekcie, rozpoczętym w 1994 roku, a dotyczącym prezentacji dynamicznych struktur danych w środowisku języka Smalltalk [112, 113].

SIAMOA

System *Siamoa* (plansza 3.9) został zrealizowany w Centre d'Automatique (obecnie Laboratoire d'Automatique I^3D) na Uniwersytecie Lille 1 przez Frédérica Van de Veire'a [115 – 121]. Jest on platformą ułatwiającą projektowanie, realizację i uruchamianie oprogramowania a także analizę działania algorytmów, działającą w środowisku języka *Smalltalk*. System obejmuje dwa podsystemy: System Wspomagania Programowania Tekstowego (*SWPT*) i System Wspomagania Programowania Wizualnego (*SWPW*). *SWPT* jest systemem animacji algorytmów zapisanych w językach *Smalltalk* lub *Pascal*. Z punktu widzenia użytkownika, *SWPT* jest rodzajem debuggera umożliwiającego wykonywanie programów w trybie nadzorowanym, połączonego z prezentacją wybranych struktur danych w formie graficznej lub tekstowej. Drugi z podsystemów, *SWPW*, umożliwia tworzenie i uruchamianie programów w sposób wizualny. Mimo że obejmuje specyficzną wizualizację tak utworzonych programów, jest to zasadniczo system programowania wizualnego, oparty o pewien przepływowy język wizualny.

W systemie *Siamoa* wizualizacja procesu wykonania algorytmu obejmuje wizualizację przetwarzanych danych oraz wizualizację postępu wykonania programu. Oba typy wizualizacji są dostępne zarówno dla programów stworzonych wizualnie, jak i tradycyjnych programów zapisanych tekstowo.

Wizualizacja postępu wykonania programu stworzonego wizualnie jest realizowana w oknie bardzo podobnym do tego, w którym programy są tworzone. Wykorzystuje się stosunkowo prosty

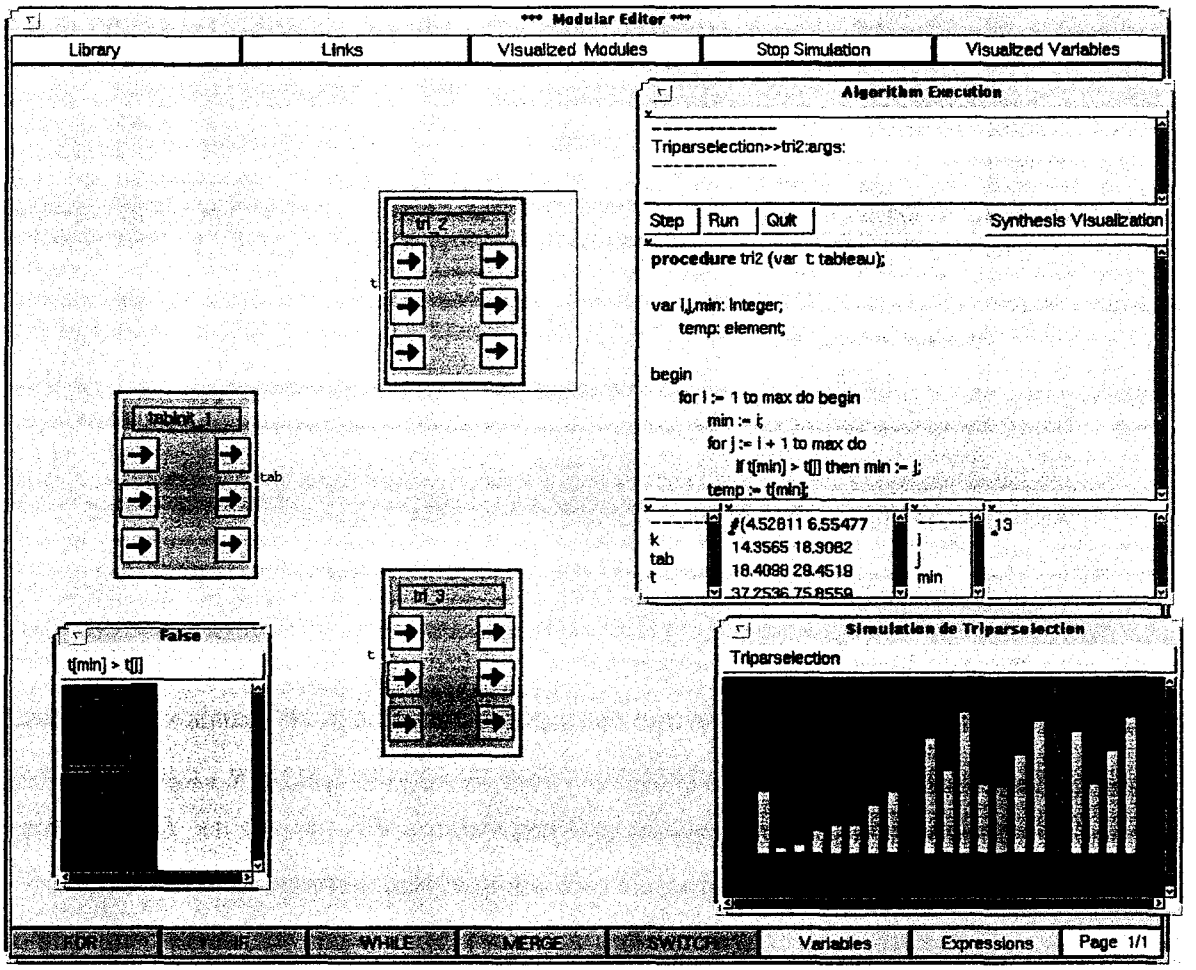
mechanizm podświetlania bloków graficznych odpowiadających modułom (fragmentom) programu, które w danej chwili są analizowane i wykonywane.

Wizualizacja postępu wykonania programu napisanego tekstowo jest realizowana w obrębie głównego okna wizualizacji, razem z projekcją reprezentacji graficznych danych. Sposób przedstawienia jest bardzo zbliżony do używanego przez standardowy debugger środowiska Smalltalk: wyświetlany jest tekst wykonywanej metody lub (podprogramu w *Pascalu*), przy czym podświetlony jest fragment tekstu odpowiadający wykonywanej operacji.

Wizualizacja danych jest dostępna i realizowana w podobny sposób dla obu typów programów: stworzonych wizualnie lub tekstowo. Wykorzystywane są takie elementy graficzne, jak linie, prostokąty, okręgi, pola tekstowe i inne. Mogą one być wykorzystywane indywidualnie lub jako składowe bardziej skomplikowanych tworów graficznych. Wartości poszczególnych zmiennych odpowiadają wybranym atrybutom obiektów, takim jak pozycja, rozmiar, kształt czy kolor. Obiekty graficzne wraz ze zbiorem określających je atrybutów stanowią **reprezentację graficzną** (franc. *représentation graphique*) danych poddawanych wizualizacji. Zmiana stanu danych jest odzwierciedlana za pomocą odpowiedniej zmiany atrybutów ich reprezentacji graficznej. Odbywa się to poprzez zastosowanie transformacji graficznej zwanej **metodą animacji** (franc. *méthôde d'animation*). Typowymi przykładami metod animacji są: przesunięcie, powiększenie, zmniejszenie, deformacja i zmiana koloru. Animacja skomplikowanych struktur danych wymaga jednak specyficznych metod animacji, nawiązujących do poszczególnych operacji dających się wykonać na tych strukturach, takich jak tworzenie lub usuwanie elementów, zamiana dwóch elementów itd.

Architektura maszyny projekcyjnej systemu wykorzystuje tzw. model *POST*: Prezentacja – Obiekt Symulacji – Translacja (franc. *Présentation, Objet de Simulation, Traduction*), stosowany uprzednio do budowy interaktywnych platform symulacji procesów przemysłowych ([31, 32], por. też rozdział 7.1). Model *POST* składa się z trzech głównych części (rys. 3.1). Są to:

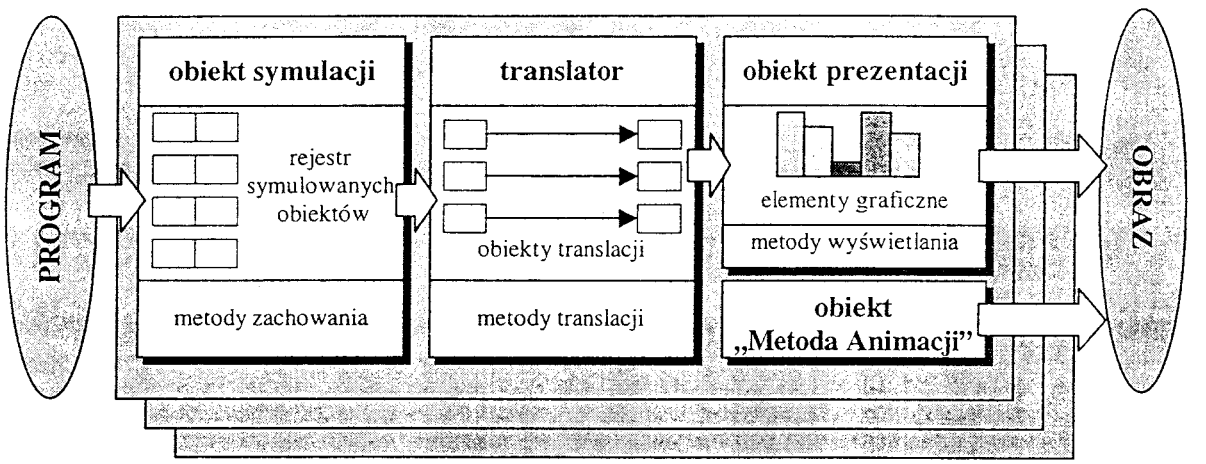
- Obiekt symulacji, reprezentujący abstrakcyjną część aplikacji. W ogólnym przypadku jest rozumiany jako kombinacja części statycznej złożonej z jednego lub kilku atrybutów oraz części dynamicznej. W przypadku systemu *Siamoa* wykorzystana jest tylko część statyczna. Jej atrybuty reprezentują zmienne wizualizowanego algorytmu.
- Obiekt prezentacji, związany z graficzną stroną systemu. Określa reprezentację graficzną i metody animacji dla poszczególnych obiektów symulacji, a także odpowiada za wyświetlanie całej projekcji.



Plansza 3.9. Typowy widok okna wizualizacji w systemie SIAMOA

- Translator, zapewniający wzajemną zgodność obszarów abstrakcyjnego i graficznego. Obiekt ten współpracuje z biblioteką metod translacji.

Konstrukcja jednostkowego modułu w oparciu o model *POST* obejmuje w systemie *Siamoa* dwa kroki. W pierwszym użytkownik wybiera obiekty symulacji (zmienne w poddawanym wizualizacji programie) i określa dla nich stosowne reprezentacje graficzne i metody animacji. Do-



Rys. 3.1. Oparta na modelu *POST* architektura systemu *Siamoa*

stępne są zarówno biblioteki gotowych reprezentacji graficznych i metod animacji, jak i edytor graficzny, pozwalający na poszerzenie istniejącego repertuaru. Drugi krok polega na stworzeniu obiektu translacji i wykorzystaniu go do połączenia utworzonych uprzednio obiektów symulacji i prezentacji. W ten sposób, gdy w czasie projekcji nastąpi zmiana wartości określonej zmiennej w wizualizowanym programie, uaktywniona zostanie odpowiednia metoda translacji, która uruchomi animację (realizowaną w obrębie warstwy prezentacji).

Wizualizacja danych podczas procesu wykonania algorytmu może przyjąć formę prostą lub syntetyczną. Pierwsza z nich uwidacznia wartości pojedynczych zmiennych, podczas gdy druga – odzwierciedla bardziej ogólne zachowanie algorytmu.

Prosta wizualizacja danych umożliwia przedstawienie wartości poszczególnych zmiennych (prostych lub złożonych) w danym momencie, a także pokazanie, jak się one zmieniają w trakcie wykonania programu. Dane są wyświetlane w formie adekwatnej do typu zmiennej: na przykład zmienne liczbowe są pokazywane w postaci wykresu z zaznaczeniem aktualnej wartości, zmienne logiczne w postaci dwustanowych semaforów, tablice – w postaci zestawu prostokątnych słupków.

Syntetyczna wizualizacja danych stanowi połączenie prezentacji kilku zmiennych (zazwyczaj zarówno prostych jak i złożonych) w obrębie pojedynczego widoku graficznego. Zmienne przedstawiane w ten sposób są zwykle semantycznie powiązane, i to właśnie te powiązania są tym, co autorzy chcieli podkreślić w tego typu wizualizacji. Przykład wizualizacji omawianego typu znajduje się w dolnym prawym rogu ekranu na planszy 3.9.

Zastosowanie modelu *POST* sprzyja tworzeniu wizualizacji charakteryzujących się wysokim poziomem abstrakcji. Dotyczy to przede wszystkim syntetycznej wizualizacji danych. Mimo jednak, że styl specyfikacji jest deklaratywny i nieinwazyjny, a system wyposażono w graficzny edytor reprezentacji, proces przygotowania projekcji jest dość uciążliwy, na pewnych etapach wymaga nawet programowania wprost w języku *Smalltalk*.

Jak już wspomniano, system *Siamoa* oferuje środki do tworzenia programów w sposób wizualny. Pokrótce przedstawimy zastosowaną metodę specyfikacji programu wizualnego, będącą adaptacją przepływowych języków wizualnych [40, 43].

Program wizualny składa się z modułów, z których każdy reprezentowany jest przez widoczny na ekranie element graficzny, dla którego określono zestaw wejściowych i wyjściowych linii danych. Użytkownik umieszcza obiekty graficzne związane z potrzebnymi mu modułami na ekranie, a następnie zaznacza połączenia pomiędzy odpowiednimi wejściami i wyjściami. W trakcie wykonania programu te moduły, dla których wszystkie dane wejściowe są ustawione, aktywizują się. Kolejność wykonania zaktywizowanych w ten sposób modułów jest nieustalona. W wyniku analizy i wykonania zaktywizowanych modułów podejmowane są związane z nimi obliczenia i generowane wyniki, a odpowiednie wyjściowe linie danych są ustawiane. To z kolei może spowodować aktywizację innych, nieaktywnych dotąd modułów. Taka propagacja obliczeń utrzymuje system w ruchu aż do momentu otrzymania końcowych wyników działania algorytmu.

3.4 SYSTEMY WIZUALIZACJI NIEIMPERATYWNYCH ASPEKTÓW OPROGRAMOWANIA

Wszystkie opisane dotąd systemy koncentrują się na wizualizacji procesu wykonania algorytmów lub programów w sposób typowy dla języków imperatywnych. Wiele systemów zostało jednak zaprojektowanych w celu wizualizacji programów napisanych w językach nieimperatywnych – funkcyjnych lub deklaratywnych, bądź skupiają się nad tymi aspektami oprogramowania, które nie są związane z imperatywnym charakterem języka, w którym zo-

stało ono napisane. W tym drugim przypadku chodzi przede wszystkim o wizualizację zależności charakterystycznych dla systemów obiektowych oraz o wizualizację związaną z problemami przetwarzania równoległego. Mimo że systemy takie są zasadniczo poza zakresem tej pracy, przytoczymy ich krótką charakterystykę, by przedstawiony tu opis systemów wizualizacji był pełny. Oczywiście rozróżnienie między „nieimperatywnymi" systemami a opisywanymi dotąd rozwiązaniami „imperatywnymi" jest płynne: na przykład w poprzednim punkcie opisaliśmy takie systemy, jak *Pavane* – przeznaczony przede wszystkim do obrazowania obliczeń równoległych, oraz *Animus* – ściśle związany z obiektowym środowiskiem *Smalltalka*.

SYSTEMY WIZUALIZACJI DLA JĘZYKÓW FUNKCYJNYCH

Kilka systemów zaprojektowano specjalnie dla języków funkcyjnych, przede wszystkim dla języka *Lisp*. Spośród nich najszerzej znany jest *KEASTLE* [133]. Wspomaga on uruchamianie programów pisanych w *Lispie* poprzez automatyczną wizualizację struktur danych tworzonych w czasie wykonywania programu. Wyświetlane obrazy można poddawać edycji, zmieniając nie tylko graficzny układ rysunku, ale także bezpośrednio manipulując przedstawianymi strukturami danych. System *KEASTLE* współpracuje z narzędziem o nazwie *FooScape*, służącym do wizualizacji struktury kodu programu. Wyświetla ono funkcje w postaci opisanych elips, połączonych strzałkami reprezentującymi wywołania. System podświetla elipsę odpowiadającą aktualnie realizowanej funkcji.

System *LogoMedia* [134] wizualizuje programy napisane w języku *Logo*. Jego mocną stroną jest intensywne wykorzystanie dźwięku.

SYSTEMY WIZUALIZACJI DLA JĘZYKÓW DEKLARATYWNYCH

System *Transparent Prolog Machine (TPM)* [135, 136], zaprojektowany na brytyjskim Otwartym Uniwersytecie (*Open University*), jest uważany za jeden z najbardziej udanych automatycznych systemów wykorzystywanych do graficznego uruchamiania programów. *TPM* udostępnia dwa podstawowe typy widoków: gruboziarnisty *CGV* (ang. *coarse-grained view*) oraz drobnoziarnisty (ang. *fine-grained view*), zwany diagramem *AORTA*. Widok gruboziarnisty *CGV* jest oparty o standardowe drzewo *i/lub*. Pokazuje całą przestrzeń wykonania programu z węzłami reprezentującymi cele. Każdy węzeł jest pokolorowany w sposób określający jego aktualny stan. Wizualizacja dużych programów jest ułatwiona dzięki możliwości przewijania obrazu i nawigacji w drzewie. Drobnoziarnisty widok *AORTA* (ang. *And/OR*

Tree, Augmented) pozwala użytkownikowi skoncentrować się na poszczególnych węzłach – celach, uzyskując szczegóły dotyczące przepływu danych. Pokazuje on hierarchię drzewa *i/lub* dla pewnego poddrzewa, w którym każdy węzeł – cel jest reprezentowany za pomocą prostokąta. W jego górnej części wyświetlany jest aktualny status celu, w dolnej zaś – liczba aktualnie rozpatrywanych warunków (klauzul). Prostokąt posiada też pewną liczbę pionowych kresek, reprezentujących warunki. Zaznaczony jest też status każdego z nich. Bieżący cel i odpowiadające mu warunki są podane obok każdego diagramu *AORTA*.

Inne systemy, jak na przykład *Graphical Environment for Monitoring Prolog Programs* Lazzeriego, również wykorzystują techniki podobne do opisanych.

Systemy wizualizacji dla architektur obiektowych

Algorytmy zaimplementowane w programach napisanych obiektowo dają się zwykle łatwo wizualizować za pomocą zwykłych systemów wizualizacji, takich jak na przykład opisane w poprzednim punkcie. Obecnie skupimy się na systemach wizualizujących te aspekty oprogramowania, które są typowe dla technik obiektowych. Obszerniejszy przegląd zagadnienia można znaleźć między innymi w [137, 139, 142].

Proste widoki semantyczne systemów obiektowych skupiają się wokół zagadnień hierarchii klas, widoczności i interfejsów. Stają się one już standardem przemysłowym, szeroko stosowanym w takich produktach komercyjnych, jak *Microsoft Visual C++* [72] (który opisaliśmy wcześniej) lub większość środowisk programowania w języku *Smalltalk*. Jednak najistotniejszym problemem w architekturach obiektowych są współzależności i współpraca pomiędzy poszczególnymi wcieleniami (instancjami) obiektów, decydująca o zachowaniu się systemu jako całości. Projektanci często dokumentują współpracę pomiędzy obiektami rysując tzw. wykresy protokołów, w których zaznacza się poszczególne wystąpienia obiektów oraz przesyły komunikatów – te ostatnie po prostu za pomocą strzałek. Animacja stanowi tu szansę uwidocznienia kolejności przesyłania komunikatów oraz cykli życiowych obiektów. Zaprojektowany przez grupę pod kierownictwem Johna Stasko system *GROOVE* [139, 142] pomaga projektować i dokumentować programy obiektowe. Jest to przede wszystkim system programowania wizualnego, ale posiada też cechy systemu wizualizacji. Generowany obraz jest bardzo podobny do konwencjonalnych, opisanych powyżej wykresów protokołów, ale styl prezentacji jest dynamiczny: użytkownik tworzy scenariusz, którego ściśle określonymi w czasie elementami są takie zdarzenia, jak tworzenie i destrukcja obiektów i wywoływanie poszczególnych metod. Są one uwidaczniane za pomocą technik animacyjnych. Scenariusz

taki stanowi swoistą formę animowanej dokumentacji; na jego podstawie jest też generowany szkieletowy program w *C++*. Program taki może być następnie wykonywany w specjalnym trybie, w którym steruje on wizualizacją. Również konwencjonalnie zaprojektowane i wykonane programy można zaadaptować do animacji za pomocą systemu *GROOVE*.

W nowszych pracach na ten temat Stasko analizuje tak zwane wzory komunikatów (ang. *message patterns*) – powtarzające się sekwencje komunikatów i konstrukcji/destrukcji obiektów [138]. Otrzymuje on fascynujące rezultaty w wyniku analizy bardzo dużej liczby komunikatów przesyłanych pomiędzy obiektami. Prototypowy system automatycznie identyfikuje wzory komunikatów i pozwala użytkownikowi na ich obserwację i modyfikację. Daje to zupełnie niezwykłe możliwości akwizycji analitycznych danych na temat dynamicznego zachowania się bardzo dużych systemów obiektowych, i porównania ich z danymi analitycznymi wynikającymi ze stawianych systemowi wymagań. Stanowi to unikalną metodę weryfikacji potężnych, rzeczywistych systemów. Analiza wzorów komunikatów wymaga zebrania danych na temat dziesiątków tysięcy indywidualnych przesyłów komunikatów. Do przetworzenia tych informacji zastosowano oryginalne techniki fresku informacyjnego (ang. *information mural*) [27] oraz zbliżenia semantycznego (ang. *semantic zoom*) [28]: obydwie pozwalają na wizualizację bardzo dużych zbiorów danych na pojedynczym obrazie.

Wydaje się, że system *GROOVE* jest systemem najbardziej zaawansowanym. Kilka innych rozwiązań jedynie krótko zasygnalizujemy. W. De Pauw, R. Helm, D. Kimelmani J. Vlissides [137] stworzyli narzędzie do wizualizacji cyklu życia obiektów, odwołań w obrębie klas i pomiędzy klasami, a także historii alokacji pamięci – wszystko to w postaci animowanego raportu generowanego *post-mortem*. H. Koike [21] wykorzystał trójwymiarowe techniki wizualizacyjne dla zobrazowania przesyłu komunikatów w złożonym systemie. *Program Explorer* [141] D. B. Lange'a i Y. Nakamury wykorzystuje widoki klas i obiektów z zaznaczeniem ich wzajemnych relacji oraz historii przesyłu komunikatów. C. Laffra i A. Malhotra stworzyli wizualny debugger dla języków *C++* i *Smalltalk*, o nazwie *HotWired* [140]. Pokazuje on indywidualne obiekty wraz z wartościami atrybutów i wprowadza prostą wizualizację dla zilustrowania przesyłu komunikatów pomiędzy nimi. System przeznaczony jest przede wszystkim do zadań związanych z uruchamianiem oprogramowania – między innymi pozwala stosunkowo łatwo odtwarzać sekwencje komunikatów, które doprowadziły do wystąpienia określonej sytuacji błędnej. Z drugiej strony wydaje się, że system ten wymaga zbyt wiele ręcznego programowania, by stać się efektywnym narzędziem inżynierskim.

Systemy wizualizacji dla obliczeń równoległych

Mimo że wiele z opisanych dotąd systemów może wizualizować programy i algorytmy równoległe, znaczna liczba systemów została zaprojektowana specjalnie dla zobrazowania specyficznych cech tego typu przetwarzania danych. Programowanie i uruchamianie programów w języku równoległym może być znacznie bardziej skomplikowane, niż w przypadku systemów sekwencyjnych, ze względu na większą liczbę wzajemnie oddziałujących na siebie elementów obliczeniowych, a także niedeterministyczny sposób wykonywania programów. Liczni badacze twierdzą, że programowanie współbieżne jest idealnym polem dla wizualizacji oprogramowania ze względu na bardzo duże ilości wysoce dynamicznych informacji.

Wizualizacja programów równoległych wiąże się z licznymi problemami specyficznymi dla tej dziedziny zastosowań, która zresztą pozostaje poza sferą naszych zainteresowań w obrębie tej pracy. Przegląd problematyki można znaleźć w [11, 93, 95, 143, 145]. Obecnie przedstawimy tylko krótki przegląd rozwiązań. Jeden z systemów, *Pavane*, został już opisany w punkcie 3.3 ze względu na swoje cechy istotne również dla prac dotyczących wizualizacji obliczeń sekwencyjnych.

System *IVE* (*Integrated Visualisation Environment*) [146] jest poświęcony wizualizacji programów masywnie równoległych. Zawiera diagramy wywołań, grafy zależności, elementy wizualizacji przepływu sterowania, a także specjalnie konstruowane abstrakcje dostosowane do konkretnych zastosowań. System *PIE* (*Parallel Programming and Instrumentation Environment*) [147] koncentruje się na automatycznej wizualizacji podstawowych elementów kodu współbieżnego (*sync*, *join*, *put*, *get* itd.). Obsługuje on szeroki zestaw języków programowania, jak *C*, *MPC*, *Ada* i *Fortran*. System *ParaGraph* [144] oferuje 25 różnych sposobów przedstawienia danych zebranych podczas wykonania programu w systemie równoległym z przesyłem komunikatów. W sposób dynamiczny uwidacznia między innymi takie szczegóły, jak użycie procesorów i zestawienia wydajności programów. System Zernicka, Snira i Malki'ego [148] również zbiera informacje podczas komunikacji między procesami. System wykorzystuje techniki wizualizacyjne do analizy błędów związanych z równoległością. System nadaje się do zastosowania w architekturach wyposażonych w tysiące procesorów w dowolnym języku, w którym *explicite* występują instrukcje komunikacji i synchronizacji.

3.5 OGÓLNA KLASYFIKACJA SYSTEMÓW – KONKLUZJA

Wiele wysiłku włożono w dokonanie klasyfikacji narzędzi wizualizacji oprogramowania. Jedną z pierwszych tego typu prób podjął B. A. Myers [3], tworząc złożoną z sześciu kategorii taksonomię. Opracował ją korzystając z dwóch osi: poziomu abstrakcji (czy system ilustruje kod, dane czy algorytm), oraz poziomu animacji (czy system jest statyczny czy dynamiczny), otrzymując kratę dwa na trzy. J. T. Stasko i C. Patterson [12] wprowadzili czterowymiarową klasyfikację obejmującą aspekt, abstrakcję, animację i automatyzację. A. Price, R. M. Baecker i I. S. Small zaproponowali bardzo dokładną taksonomię [5], obejmującą aż 47 kategorii, zgrupowanych w sześciu kategoriach nadrzędnych: zasięgu, treści, formy, metody, interakcji i efektywności. Ellershow i Oudshoorn prezentują odmienne podejście do problemu: klasyfikują systemy względem sześciu paradygmatów języków programowania (imperatywne, równoległe, funkcyjne, obiektowe, deklaratywne oraz języki baz danych z obiektami trwałymi), oraz względem trzech architektur systemów (jednoprocesorowe, wieloprocesorowe i rozproszone heterogeniczne sieci komputerowe).

Nie mamy w tym miejscu ambicji zaproponowania nowej, kompletnej klasyfikacji. Dokonamy jedynie przeglądu systemów względem dwóch kategorii, które uważamy za najistotniejsze z punktu widzenia tej pracy. Pierwsza z nich to **poziom abstrakcji** generowanych przez system projekcji. Animacje algorytmów charakteryzujące się wysokim poziomem abstrakcji mogą znacznie ułatwić zrozumienie sposobu działania poddawanego wizualizacji oprogramowania. Druga z analizowanych kategorii to **poziom trudności przygotowania projekcji**. Zależy on w ogromnym stopniu od zastosowanej w systemie metody wizualizacji i rodzaju specyfikacji projekcji. W naszym przeglądzie pominiemy systemy wizualizacji nieimperatywnych aspektów oprogramowania, opisane w punkcie 3.4.

Najniższy poziom abstrakcji reprezentują narzędzia wizualizacji programów. Należą do nich: system wizualizacji kodu źródłowego *SEE*, system wizualizacji struktur danych *Incense/Pascal Genie*, oraz *Microsoft Visual C++*, zawierający rozbudowane narzędzia wizualizacji struktur danych, organizacji kodu źródłowego oraz stanu programu w czasie wykonania. Mimo niskiego poziomu abstrakcji, systemy te świetnie spełniają swoją rolę. Dowodem tego jest nie tylko sukces komercyjny *Microsoft Visual C++*, ale także licznych systemów innych firm, które udostępniają podobne możliwości. O sukcesie przesądził w dużym stopniu całkowicie automatyczny tryb generowania wizualizacji, nie wymagający wysiłku ze strony użytkownika.

Projekcje tworzone za pomocą systemów wizualizacji algorytmów z definicji charakteryzują się wyższym poziomem abstrakcji, niż to miało miejsce w przypadku narzędzi wizualizacji programów. I tak, abstrakcyjne projekcje są możliwe w systemie *ANIM*, ale zasadniczo muszą być tworzone ręcznie. Możliwości systemu *Animus* są pod wieloma względami ograniczone; trudno określić, jaki jest możliwy do osiągnięcia poziom abstrakcji w przypadku wizualizacji innych algorytmów, niż sortowanie. *UWPI* umożliwia tworzenie wysoce abstrakcyjnych widoków danych, ale nie pozwala na wprowadzanie własnych abstrakcji. Możliwe jest tylko zastosowanie reprezentacji przewidzianych przez twórców systemu. Zapewne najwyższy poziom abstrakcji zapewniają systemy: *Balsa/Zeus*, *XTango* i *Pavane*, oraz nieco im ustępujące *SANAL*, *Whizz*/Witness, *WinSANAL* i *Siamoa*. Teoretycznie najwyższy poziom abstrakcji można by uzyskać w przypadku filmów animowanych, takich, jak *Sorting Out Sorting*. Ograniczeniem jest tu tylko inwencja twórcy filmu – choć w praktyce nakładają się tu oczywiście też ograniczenia techniczne, finansowe i czasowe (przypomnijmy, że wspomniany klasyczny film Baeckera powstawał cztery lata).

Wśród systemów wizualizacji i animacji algorytmów niemal regułą jest wyższy poziom trudności przygotowania projekcji, niż to miało miejsce w przypadku narzędzi wizualizacji programów. Najżmudniejsze jest przygotowanie filmu animowanego, oczywiście tylko przy założeniu, że nie korzystamy przy tym z żadnego systemu komputerowego (założenie to raczej nie bywa już w obecnych czasach spełnione). System *ANIM* wymaga konstrukcji wizualizacji za pomocą bardzo prostych operacji graficznych – co jest niewątpliwie skuteczną metodą w przypadku relatywnie nieskomplikowanych obrazów (potwierdzają to badania empiryczne), może jednak być bardzo żmudne, gdy celem jest projekcja wysoce abstrakcyjna i zaawansowana technicznie. Przygotowanie wizualizacji w systemach sterowanych zdarzeniami, w których dominuje imperatywny styl specyfikacji wizualizacji, jest żmudne, a tworzenie wysoce abstrakcyjnych widoków może się wiązać z daleko idącymi przeróbkami tekstu źródłowego programu. Z drugiej strony należy zauważyć, że właśnie do tej kategorii należą klasyczne, powszechnie znane systemy, *Balsa/Zeus* i *XTango/Polka* (a także mniej znany *SANAL*) – które umożliwiają tworzenie wizualizacji odznaczających się bardzo wysokim poziomem abstrakcji. Zauważmy także, że system *XTango/Polka* posiada elementy deklaratywnego stylu specyfikacji i narzędzia graficzne ułatwiające przygotowanie projekcji. Systemy z w pełni deklaratywnym stylem specyfikacji i wizualizacją sterowaną danymi charakteryzują się niższym poziomem trudności przygotowania projekcji. Co prawda pierwszy tego rodzaju

system, za jaki jest uważany *Animus*, nie był w pełni udany, ale już *Pavane* należy do ścisłej czołówki systemów oferujących wysoce abstrakcyjne wizualizacje. Z innych systemów należących do tej kategorii wymienimy *Whizz*/Witness, *WinSANAL* i *Siamoa*. Na szczególną uwagę zasługuje system *UWPI*. Umożliwia on automatyczne generowanie graficznych reprezentacji struktur danych, które być może nie cechują się tak wysokim poziomem abstrakcji, jak, powiedzmy, projekcje w systemach *Zeus* czy *Pavane*, niemniej *UWPI* zdecydowanie przewyższa automatyczne systemy wizualizacji programów. System *UWPI* w istotny sposób odróżnia się od wszystkich innych systemów możliwością **automatycznego tworzenia abstrakcji** danych. Price, Baecker i Small uważają tę możliwość za jedną z 47 kategorii swojej klasyfikacji, nazywając ją **inteligencją systemu**.

Oczywistą konkluzją jest, że im wyższy poziom abstrakcji jest uzyskiwany w ramach projekcji, tym trudniejsza jest ona do przygotowania. Z drugiej strony olbrzymi wpływ na poziom trudności ma też sposób organizacji systemu wizualizacji. W zasadzie prawie wszystkie systemy animacji algorytmów pozwalają na tworzenie projekcji o zbliżonym poziomie abstrakcji, lecz stopień trudności ich przygotowania jest bardzo różny. Szczególnie widoczne jest, że systemy z deklaratywnym stylem specyfikacji są łatwiejsze w obsłudze, a tworzone przy ich pomocy projekcje nie muszą wcale ustępować tym tworzonym za pomocą systemów imperatywnych.

Tym, czego zasadniczo brakuje systemom wizualizacji oprogramowania, jest wspomniana wyżej cecha nazwana **inteligencją**. Systemy animacji algorytmów wymagają, by użytkownik samodzielnie tworzył abstrakcje, czy też wprowadzał do projekcji zawartość intencyjną. Systemy automatyczne nie dają takiej możliwości, i generowane przez nie projekcje odznaczają się bardzo niskim poziomem abstrakcji. Wyjątkiem jest system *UWPI*, który mimo swych ograniczeń, w sposób automatyczny generuje dość zaawansowane abstrakcje struktur danych. Właściwie jest to jedyny z omawianych systemów, w przypadku którego możemy mówić o inteligencji.

Celem naszej pracy jest stworzenie metody animacji algorytmów która również odznacza się pewnym poziomem inteligencji. Rozumiemy przez to, że korzystający z niej system może wspomagać tworzenie abstrakcyjnych widoków algorytmu bez udziału i bez wysiłku ze strony wizualizatora, w sposób automatyczny.

4
ANIMACJA ALGORYTMÓW OPARTA NA ŚLEDZENIU PRZEPŁYWU DANYCH

Niniejszy rozdział jest poświęcony metodzie animacji algorytmów opartej na śledzeniu przepływu danych. W naszym przeświadczeniu jest to najbardziej oryginalny i najważniejszy wkład, jaki praca wnosi do rozwoju dziedziny wizualizacji algorytmów.

W początkowej części rozdziału przedstawione zostaną różne poziomy opisu procesu wykonania algorytmu, wyjaśnimy też pryncypialną rolę przepływu danych. Poszukując modelu wizualizacji, który pozwala wyekstrahować z poddawanego wizualizacji oprogramowania te informacje, które są najistotniejsze z punktu widzenia obserwatora, sformułowano trzy wymagania stawiane skutecznej wizualizacji danych. Są to:

- Postulat obrazowania przepływu danych.
- Postulat przestrzennej redukcji informacji, wykluczającej z projekcji nieistotną część informacji dotyczącej przestrzennego układu przepływu danych.
- Postulat temporalnej redukcji informacji, wykluczającej z projekcji nieistotną część informacji dotyczącej uporządkowania przepływu danych w czasie.

W kolejnych podrozdziałach przedstawimy zasadniczy model działania algorytmu sformułowany w terminach przepływu danych, a także nasz algorytm animacji algorytmów, który uwzględnia wszystkie trzy przytoczone wyżej postulaty. Algorytm, a w każdym razie jego bardziej zaawansowane wersje, opisano korzystając z formalizmu sieci Petriego [149].

Zastosowanie przedstawionego algorytmu w systemie *Daphnis* pozwoliło na zautomatyzowanie części procesu przygotowania wizualizacji, która w istniejących dotychczas systemach wymagała ręcznego projektowania i strojenia. Szczegóły dotyczące implementacji algorytmu w systemie *Daphnis* przedstawiono w rozdziale 7.2.

4.1 POZIOMY OPISU ALGORYTMÓW

Proces wykonania algorytmu, a ściślej – wykonania programu realizującego ten algorytm – może być rozpatrywany na różnych poziomach. Najbardziej naturalny jest model wywiedziony ze świata rzeczywistego, odwołujący się do realnie istniejących obiektów, które pełnią funkcję metafor, ilustrujących zjawiska występujące na poziomie algorytmu. Przykładem może być przedstawienie operacji zamiany wartości pomiędzy dwiema zmiennymi za pomocą dwóch kul, poruszających się synchronicznie po łukowatych torach tak, by w efekcie każda z kul zajęła pozycję pierwotnie zajmowaną przez drugą z nich (plansza 4.1a). Zauważmy, że synchroniczny charakter tego ruchu jest niezmiernie istotny, pozwala bowiem uniknąć kolizji. Nie możemy umieścić żadnej z kul w docelowym położeniu, dopóki kula znajdująca się tam na początku nie zostanie usunięta.

W porównaniu z metaforą świata rzeczywistego, łatwą do wyrażenia za pomocą języka naturalnego, opis algorytmu za pomocą typowego, formalnego języka programowania często powoduje konieczność wprowadzenia dodatkowych elementów, wymuszonych przez specyfikę użytego języka czy środowiska. Częstym problemem jest niemożność wyrażenia wprost działań, które łatwo dałyby się opisać jako proces synchroniczny. Znowu dobrym przykładem jest zamiana dwóch zmiennych: zastosowanie typowego języka programowania wymaga zastosowania dodatkowej zmiennej pomocniczej. Zapis w języku C (por. plansza 4.1b) jest więc następujący:

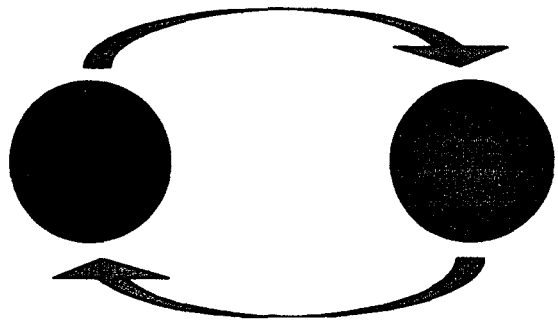
```
aux = a;
a = b;
b = aux;
```

Schodząc na niższy poziom, przeanalizujemy skompilowany kod programu. Dla zmiennych całkowitych, kod wygenerowany dla procesorów rodziny Intel 80x86 wygląda następująco[2]:

```
mov AX, a       ; prześlij a do aux poprzez rejestr AX
mov aux, AX
mov AX, b       ; prześlij b do a poprzez rejestr AX
mov a, AX
mov AX, aux     ; prześlij aux do b poprzez rejestr AX
mov b, AX
```

Ze względu na niedostępność przesyłu typu pamięć – pamięć, wykorzystano dodatkowo rejestr procesora AX. W tym kontekście zastosowanie zmiennej pomocniczej wydaje się nieuzasadnione; zamiast tego można użyć jeszcze jednego rejestru:

[2] Kod ten został skompilowany za pomocą kompilatora Microsoft™ Visual C++®, wersja 1.52.

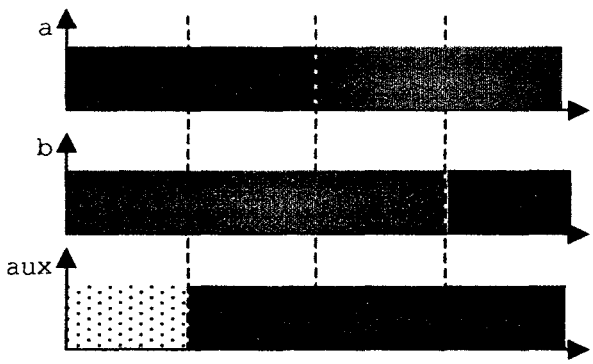


a

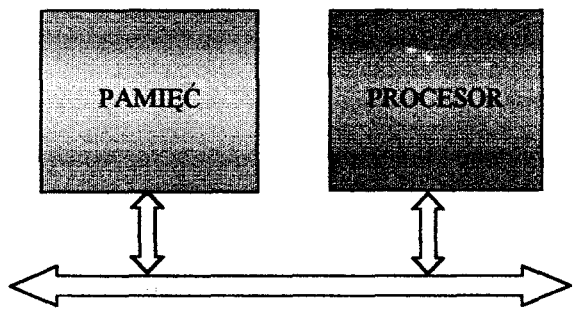
```
aux = a;
a = b;
b = aux;

mov   AX, a
mov   BX, b
mov   a, BX
mov   b, AX
```

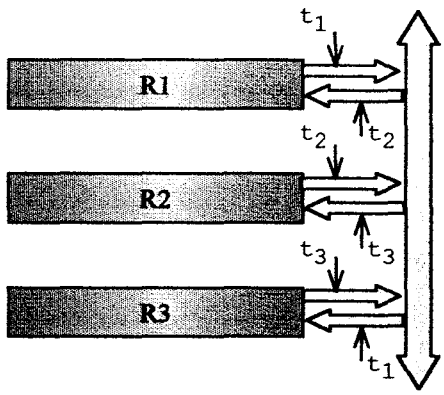
b



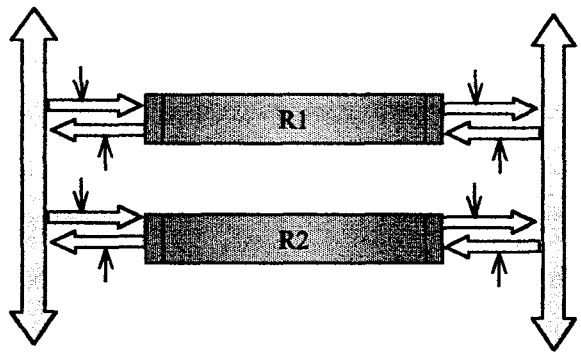
c



d



e



f

Plansza 4.1 Różne poziomy opisu operacji zamiany dwóch wartości

```
mov AX, a       ; załaduj zmienne do rejestrów
mov BX, b
mov a, BX       ; zachowaj stare wartości zmiennych
mov b, AX       ;   pod zamienionymi adresami
```

Schodząc na jeszcze niższy poziom, możemy analizować chwilowe wartości zmiennych (plansza 4.1c). Jest to poziom, na którym pracują tradycyjne debuggery. Proces wykonania algorytmu powoduje serię dyskretnych zmian wartości przechowywanych w obserwowanych strukturach danych. Możemy obserwować, jak zmieniają się wartości poszczególnych zmiennych. Na tej podstawie możemy ocenić, czy wykonanie określonych operacji zakończyło się sukcesem. Nie możemy jednak udowodnić poprawności rozwiązania, gdyż nie możemy określić, jakie związki przyczynowo-skutkowe występują pomiędzy poszczególnymi zmianami wartości, nie odnosząc się do tekstu programu (źródłowego lub skompilowanego). Znacznie większą wartość analityczną mogłaby mieć obserwacja na poziomie układów elektronicznych, z których zbudowane są współczesne komputery. Przesyły danych między pamięcią i procesorem (plansza 4.1d), lub, w prostszym przypadku, pomiędzy rejestrami (plansza 4.1e), wskazują na przyczynowo-skutkowe związki pomiędzy poszczególnymi zmianami wartości, a zatem lepiej odzwierciedlają semantykę algorytmu. Wracając do naszego przykładu z zamianą wartości dwóch zmiennych, można stwierdzić, że na tym poziomie analizy dość łatwo jest sobie wyobrazić synchroniczne rozwiązanie problemu, działające podobnie, jak w modelu wziętym ze świata rzeczywistego. Stosując dwa rejestry i dwie magistrale (plansza 4.1f) unikamy konieczności stosowania dodatkowej zmiennej pomocniczej (chociaż, z technicznego punktu widzenia, synchroniczny przesył danych po magistralach wymagałby dodania do stopnia wyjściowego rejestru typu zatrzask, dla uniknięcia efektu hazardu dynamicznego).

Na różnych poziomach opisu natrafiliśmy na różne możliwe rozwiązania i podejścia do tego samego problemu. I tak, rozwiązanie synchroniczne, naturalne w świecie rzeczywistym, nie jest możliwe do zrealizowania w klasycznym, sekwencyjnym lub asynchronicznym języku programowania; z kolei zmienna pomocnicza, niezbędna w rozwiązaniu napisanym w języku C, staje się nadmiarowa w zoptymalizowanym kodzie asemblerowym.

4.2 TRZY WYMAGANIA STAWIANE SKUTECZNEJ ANIMACJI ALGORYTMÓW

Bez cienia wątpliwości, ze wszystkich poziomów opisu algorytmów, które przytoczyliśmy w poprzednim podrozdziale, poziom obserwacji chwilowych wartości zmiennych lub reje-

strów ma najmniejszą wartość poznawczą. Z drugiej strony, obserwacja taka jest technicznie łatwiejsza do realizacji, i z tego właśnie powodu tradycyjne narzędzia do uruchamiania programów (debuggery) pracują na tym właśnie poziomie. Oczywiście nie stanowi on dobrego punktu wyjściowego dla wizualizacji algorytmów: można co prawda użytkownikowi pokazać, *co się dzieje*, ale nie można mu wyjaśnić, *dlaczego tak się dzieje*. Wizualizacje, choć stosunkowo łatwe do uzyskania – wystarczy sekwencyjnie odczytywać chwilowe wartości zmiennych i prezentować je w odpowiedniej formie graficznej – nie ułatwiają dostatecznie zrozumienia algorytmu. Dyskretne zmiany wartości zmiennych mogą być odwzorowane jako dyskretne, nagłe zmiany reprezentacji graficznej. Takie zmiany, zwykle trudne nawet do zauważenia, nie dostarczają użytkownikowi informacji na temat przyczyny zmiany wartości, ani też źródła tej nowej wartości.

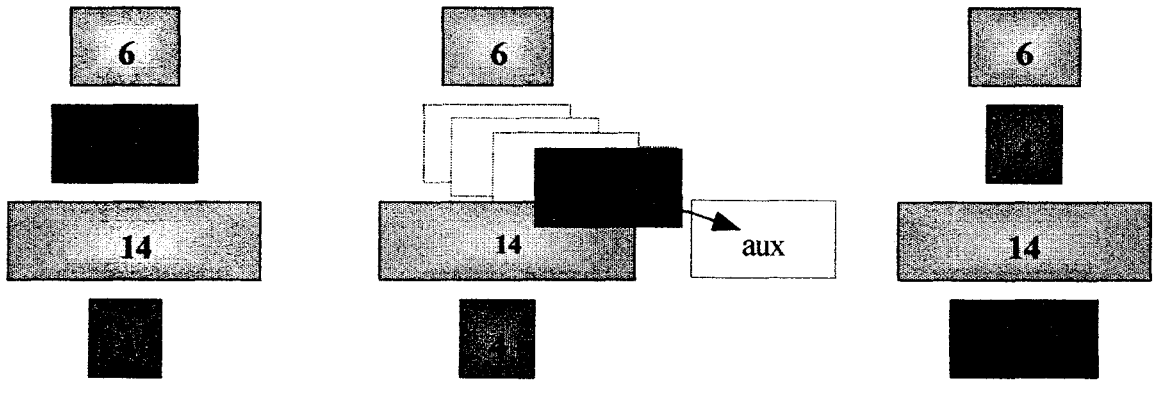
W przykładowej wizualizacji operacji zamiany wartości dwóch zmiennych zmienne te są odwzorowane w postaci prostokątów, których szerokość odpowiada wartości zmiennej (plansza 4.2). Dyskretna wizualizacja, taka jak opisana powyżej, sprowadzałaby się do nagłej zmiany szerokości odpowiednich prostokątów (plansza 4.2a).

Poszukując lepszego modelu dla animacji algorytmów, sformułowaliśmy podstawowy wymóg, jakim jest obrazowanie przepływu danych; rozwijając to podejście, doszliśmy jeszcze do dwóch innych wymagań, które spełniać powinna dobra wizualizacja, znacznie ułatwiająca zrozumienie algorytmu.

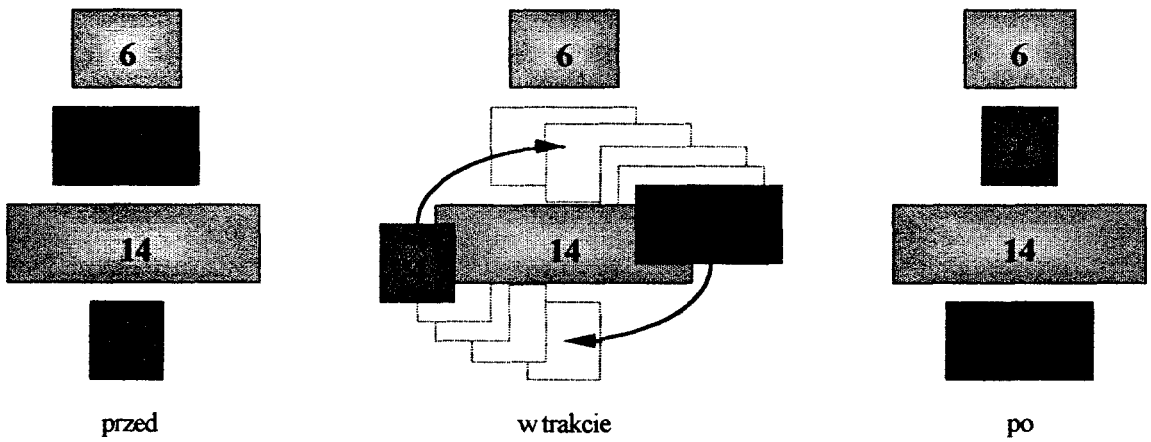
4.2.1 OBRAZOWANIE PRZEPŁYWU DANYCH

Wychwycenie związków przyczynowo-skutkowych, zachodzących pomiędzy kolejnymi, chwilowymi stanami algorytmu, jest podstawowym warunkiem zrozumienia całego algorytmu. Wizualizacja skutecznie ułatwiająca zrozumienie algorytmu musi odpowiadać nie tylko na pytanie *jak*, ale również *dlaczego*. Jeśli założymy, że chwilowy stan algorytmu wyraża się zbiorem wartości wszystkich zmiennych w danej chwili, wówczas związki przyczynowo-skutkowe można wyjaśnić **przepływami danych** pomiędzy tymi zmiennymi. W przypadku analizy algorytmu na poziomie układów elektronicznych, wynika to wprost z fizycznej zasady działania komputerów: zawartość rejestrów (zmiennych) jest konsekwencją przesyłów międzyrejestrowych. Podobnie ma się z modelami działania algorytmów nawiązującymi do świata rzeczywistego. Metaforą przepływu danych w języku C jest operator przypisania, podobnie jak instrukcja *mov* w językach asemblerowych. Istoty przetwarzania danych nie sta-

a) wizualizacja statyczna



b) płynna animacja – obrazowanie przepływu danych



c) płynna animacja, bez użycia zmiennej pomocniczej

Plansza 4.2. Przykłady wizualizacji operacji zamiany wartości dwóch zmiennych

nowi ciąg dyskretnych zmian zawartości struktur danych, ale proces ciągłego, uporządkowanego przepływu danych, w trakcie którego dane te są poddawane modyfikacjom. W tym ujęciu dane przepływające pomiędzy strukturami danych możemy traktować jak elementy **trwałe**, zmieniające zarówno parametry związane z ich wartością, jak i z pozycją (adresem), zachowujące jednak swoją tożsamość.

Dobra wizualizacja nie powinna się ograniczać do przedstawienia stanów, które program osiąga podczas wykonania, ale także ilustrować przejścia pomiędzy tymi stanami. Może to zapewnić wizualizacja, w której obiekty graficzne reprezentujące zawartość struktur danych są animowane w sposób płynny – przesuwane od pozycji reprezentującej źródło przepływu do pozycji reprezentującej miejsce przeznaczenia. Przechodzą one przez szereg pozycji pośrednich, które nie mają odniesienia do jakichkolwiek stanów, w których znajduje się program podczas wykonania, a które byłyby wykrywalne przy użyciu konwencjonalnych narzędzi uruchamiających. Obiekty graficzne nie reprezentują zatem zmiennych, ale raczej dane, przepływające pomiędzy elementarnymi strukturami danych.

PRZYKŁAD 4.1

Wizualizacja operacji zamiany wartości dwóch zmiennych, uwzględniająca postulat obrazowania przepływu danych, polega na płynnym i ciągłym przesuwaniu prostokątów reprezentujących zmienne, odzwierciedlającym ciąg trzech operacji przypisania: od *a* do zmiennej pomocniczej, od *b* do *a* i, wreszcie, od zmiennej pomocniczej do *b* (plansza 4.2b). Łatwo zauważyć, skąd i dokąd przepływają dane. Żaden z prostokątów nie zmienia rozmiarów: odzwierciedla to fakt, że istotą operacji jest przeniesienie danych pomiędzy zmiennymi *a* i *b*, nie zaś jakiekolwiek modyfikacje danych (chociaż wartości poszczególnych zmiennych ulegają w tym czasie zmianom).

4.2.2 PRZESTRZENNA REDUKCJA INFORMACJI

Istotnym wymogiem stawianym wizualizacji algorytmów jest możliwość abstrahowania w końcowej projekcji od tych elementów algorytmu, które nie są w danej chwili ważne dla użytkownika – obserwatora pokazu. Nadmiar informacji o szczegółach wykonywania pewnych operacji niepotrzebnie komplikuje projekcję, utrudniając jej zrozumienie i wychwycenie informacji istotnych. W przypadku wizualizacji ilustrującej przepływ danych, narzuca się możliwość usunięcia z projekcji zmiennych nieistotnych z punktu widzenia użytkownika, a także operacji przepływu danych dotyczących tych zmiennych. Ponieważ prowadzi to do

uproszczenia wizualizacji w sensie jej układu planarnego (lub przestrzennego) na ekranie, poprzez wyeliminowanie niektórych torów, którymi podążać mogą elementy graficzne, nazwaliśmy tę operację **przestrzenną redukcją informacji**.

Wizualizacja spełniająca postulat przestrzennej redukcji informacji to taka wizualizacja, w której ilustruje się realizację algorytmu odmienną niż faktycznie zastosowana w programie poddawanym wizualizacji, przy czym w ramach tej realizacji stosuje się mniejszą liczbę zmiennych i operacji przepływu danych. Eliminacji podlegają te elementy, które nie ułatwiają w istotny sposób zrozumienia algorytmu; są to po prostu te zmienne, które nie zostały wyspecyfikowane jako istotne przez projektanta wizualizacji. Proste ich usunięcie prowadziłoby jednak do utraty spójności całościowego obrazu przepływu danych w programie. Często bowiem zmienne uznane za nieistotne stanowią tylko etap rozleglejszych przepływów danych, rozpoczynających się i kończących w obrębie obszaru zainteresowania użytkownika. Spełnienie postulatu przestrzennej redukcji informacji wiąże się zatem z wymogiem rekonstrukcji rozerwanych ciągów przepływów danych. Elementarne operacje przepływu, których zmienne źródłowe lub docelowe znajdują się poza obszarem zainteresowania, powinny być sklejane tak, by, o ile to możliwe, formować rozleglejsze przepływy, których obydwa wierzchołki stanowią zmienne interesujące dla użytkownika.

PRZYKŁAD 4.2

Raz jeszcze posłużymy się przykładem wizualizacji zamiany wartości dwóch zmiennych. W najprostszym przypadku obejmuje on trzy zmienne, w tym jedną pomocniczą. Znacznie bliższy intuicji jest jednak model operacji zaczerpnięty ze świata rzeczywistego, który nie uwzględnia trzeciej zmiennej. Wizualizacja polegać może więc na przedstawieniu jedynie dwóch operacji przepływu danych: od *a* do *b* i od *b* do *a* (plansza 4.2c). Jeśli obserwator projekcji chciałby poznać szczegółowo sposób zamiany wartości dwóch zmiennych w określonym języku programowania, takie uproszczenie nie byłoby – oczywiście – wskazane. W większości przypadków jednak operacja zamiany wartości stanowi fragment obszerniejszego algorytmu, na przykład sortowania. Wówczas szczegóły dotyczące zmiennej pomocniczej i ciągu przypisań nie są istotne dla większości użytkowników, i niepotrzebnie komplikują obraz, nie ułatwiając zrozumienia algorytmu. Decyzję, czy poddawać wizualizacji zmienną pomocniczą i dodatkowe operacje przepływu danych, czy też nie, podejmuje użytkownik systemu – projektant wizualizacji.

Zauważmy, że proste usunięcie trzeciej zmiennej prowadziłoby do wizualizacji przebiegającej według następującego scenariusza:

1. Wartość zmiennej *a* wypływa do obszaru nieważnego (bez widocznego skutku).
2. Element reprezentujący zmienną *b* przesuwa się do zmiennej *a*.
3. Element reprezentujący zmienną *b* jest kształtowany na podstawie operacji przepływu wychodzącej z obszaru nieważnego (bez widocznej przyczyny).

Pojawia się zatem konieczność sklejenia przepływów (1) i (3) i nadania im postaci pojedynczej operacji przepływu wartości od zmiennej *a* do zmiennej *b*.

4.2.3 TEMPORALNA REDUKCJA INFORMACJI

Temporalna redukcja informacji polega na usunięciu z pokazu części informacji dotyczącej czasowego następstwa elementarnych operacji przepływu danych. W szczególności, pewne operacje, w rzeczywistości wykonywane sekwencyjnie, są obrazowane tak, jakby były realizowane równolegle i synchronicznie. Tak więc przedstawiana realizacja algorytmu różni się od faktycznie użytej w poddawanym wizualizacji programie.

Temporalna redukcja informacji bezpośrednio wiąże się z postulatem redukcji przestrzennej. Sklejanie elementarnych przepływów celem uzyskania przepływu bardziej rozległego siłą rzeczy narusza sposób uporządkowania tych przepływów. Elementarne przesyły należące do różnych zgrupowanych przepływów mogą się bowiem wzajemnie przeplatać. Ich współbieżne przedstawienie jest wówczas koniecznym wymogiem technicznym. Z drugiej jednak strony staramy się wprowadzać redukcję temporalną również i wtedy, gdy nie jest ona narzucona takimi wymogami. Jesteśmy przekonani, że może to istotnie wzbogacić wizualizację i pomaga użytkownikowi lepiej zrozumieć algorytm. Pewien stopień synchronicznej równoległości jest bowiem naturalny i typowy dla sposobu rozumowania człowieka (czyli ludzkiego systemu kognitywnego). Zachowujemy się i działamy na ogół współbieżnie. Przekonać o tym może niewielki eksperyment: wystarczy poprosić kogoś o zamienienie miejscami dwóch niewielkich przedmiotów. Większość ludzi wykona tę czynność obiema rękami, w sposób współbieżny i synchroniczny.

Szczegółowe informacje o następstwie czasowym operacji, które są niezależne od siebie i są wykonywane w krótkim odstępie czasu, częstokroć nie tylko nie ułatwiają zrozumienia istoty algorytmu, ale wręcz odwrotnie – odwracają uwagę obserwatora od innych, ważniejszych elementów projekcji.

PRZYKŁAD 4.3

Wynikające z potrzeby uwzględnienia postulatu redukcji przestrzennej sklejenie dwóch elementarnych przepływów danych w ramach wizualizacji zamiany wartości dwóch zmien-

nych (plansza 4.2c) stwarza potrzebę pewnego zrównoleglenia pokazywanych operacji, gdyż trzeci z przepływów rozpocząć się musi po rozpoczęciu sklejonego przepływu, a zakończyć przed jego zakończeniem. Może to prowadzić do następującego scenariusza:

1. Element reprezentujący zmienną *a* rozpoczyna ruch w kierunku zmiennej *b*.
2. Element reprezentujący zmienną *b* przesuwa się do zmiennej *a*.
3. Element reprezentujący zmienną *a* kończy swój ruch i jest umieszczany w miejscu pierwotnie zajmowanym przez zmienną *b*.

Znacznie lepszym opisem tej operacji jest jednak symetryczne, w pełni synchroniczne, równoczesne przypisanie *a* do *b* i *b* do *a*. Informacja na temat czasowego następstwa przypisań nie jest na ogół istotna, szczególnie, że istnieją dwa różne poprawne sposoby uporządkowania. W pełni równoczesne i symetryczne przedstawienie lepiej odzwierciedla istotę operacji, w której żadna ze zmiennych nie jest wyróżniona.

4.2.4 WYMAGANIA STAWIANE ANIMACJI ALGORYTMÓW – PODSUMOWANIE

Zbierzmy raz jeszcze nasze trzy wymagania stawiane skutecznej wizualizacji algorytmów:

- **Postulat obrazowania przepływu danych**: wizualizacja powinna wykorzystywać płynną animację w celu przedstawienia operacji przepływu danych, jakie mają miejsce podczas procesu wykonania algorytmu.
- **Postulat przestrzennej redukcji informacji**: wizualizacja powinna wykluczyć z projekcji informacje dotyczące nieistotnych zmiennych, zachowując przy tym całościowy obraz przepływu danych w algorytmie.
- **Postulat temporalnej redukcji informacji**: wizualizacja powinna zrównoleglić wykonanie niektórych przedstawianych operacji elementarnych.

Zastosowanie powyższych postulatów powoduje podwyższenie poziomu abstrakcji wizualizacji. W rozdziale 2.2 podano trzy wyróżniki wizualizacji cechującej się wysokim poziomem abstrakcji. Są to:

1. Abstrahowanie od elementów nieistotnych lub mało istotnych,
2. Zmiana realizacji algorytmu,
3. Wprowadzenie zawartości intencyjnej (ang. *intention contents*), czyli informacji semantycznych ułatwiających zrozumienie programu, nie występujących *a priori* w jego tekście, a reprezentujących wiedzę projektanta na temat tego programu.

Oba postulaty redukcji informacji – przestrzennej i temporalnej – czynią zadość warunkowi (1). Proponowane wizualizacje przedstawiają równoległą, synchroniczną realizację algorytmu, korzystającą z mniejszej liczby zmiennych, tym samym spełniają warunek (2). Obrazowane przepływy danych wprowadzają do projekcji elementy, które nie dają się wywnioskować z samej tylko obserwacji chwilowych zawartości struktur danych; co więcej, elementy graficznej reprezentacji danych przechodzą w trakcie płynnej animacji przez stany pośrednie, które w ogóle nie mają odpowiednika w wykonywanym programie, ale stanowią użyteczne narzędzie ułatwiające zrozumienie algorytmu; tym samym spełniony zostaje warunek (3). Zauważmy, że w tym przypadku *zawartość intencyjna* nie reprezentuje wiedzy projektanta, ale jest raczej wyrazem inteligencji systemu, który posiada pewien poziom wiedzy na temat poddawanego wizualizacji oprogramowania.

Wizualizacje spełniające wszystkie trzy postulaty cechują się zatem stosunkowo wysokim poziomem abstrakcji. Większość istniejących wizualizacji algorytmów dąży do ich spełnienia; przykładem mogą być liczne, bo stworzone chyba we wszystkich istniejących systemach animacje algorytmów sortowania: operacje elementarnej zamiany wartości dwóch zmiennych niemal niezmiennie są ilustrowane jako proces symetryczny i synchroniczny, bez użycia zmiennej pomocniczej. Dotychczas jednak wymagało to świadomej decyzji projektanta i ręcznego projektowania. Proponowany przez nas algorytm animacji algorytmów, który niżej przedstawiamy, umożliwia otrzymywanie takich wizualizacji w sposób automatyczny.

4.3 OPIS PROCESU WYKONANIA ALGORYTMU Z ZASTOSOWANIEM POJĘCIA PRZEPŁYWU DANYCH

Proces wykonania algorytmu, czy ściślej, wykonania programu ten algorytm realizującego, może być skutecznie opisany jako sekwencja operacji przepływu danych. Nie należy tego mylić z opisem samego algorytmu; nasz model określa jedynie, co się dzieje podczas wykonania programu i nie może być użyty jako narzędzie do specyfikacji algorytmu.

Pojęcie **zmiennej** jest krytyczne dla naszego modelu. Jest ono zdefiniowane jako związek nazwy, typu, oraz – opcjonalnie – wartości. Nazwa zmiennej (identyfikator) pozwala na jej identyfikację (np. znalezienie jej adresu w pamięci), podczas gdy typ określa zbiór wartości dozwolonych dla tej zmiennej. Zmienną, która posiada wartość, nazywamy **zwartościowaną** (ang. *valuated*), zaś **wartościowaniem** (ang. *valuating*) nazywamy akt nadania zmiennej (nowej) wartości. Zazwyczaj zmienne nie posiadają wartości przed swoim pierwszym warto-

ściowaniem, wyjątkiem są zmienne przechowujące parametry wejściowe. Innymi słowy, zmienne zdefiniowane ale nie zainicjalizowane są uważane za niezwartościowane, mimo że mogą one (na ogół) być odczytywane i z punktu widzenia składni i semantyki języka mają wartość, będącą najczęściej przypadkowym wzorem bitów.

Na wartości zmiennych wpływ mają **operacje przepływu danych**. Operacja taka jest związana z określoną zmienną docelową oraz ze zbiorem zmiennych źródłowych (zbiór ten może być pusty). Polega ona na wartościowaniu zmiennej docelowej za pomocą wartości będącej funkcją wartości zmiennych źródłowych, przy czym wartości zmiennych źródłowych nie ulegają zmianie (chyba, że któraś z nich jest jednocześnie zmienną docelową).

Jeśli zbiór zmiennych źródłowych jest pusty, wówczas mamy do czynienia z wartościowaniem zmiennej docelowej za pomocą wartości danej *a priori* (odpowiednik adresowania natychmiastowego, np. $a = 5$). Jeśli żadna ze zmiennych źródłowych nie jest zwartościowana, również przyjmujemy wartościowanie wartością *a priori* (obserwowalną po zakończeniu operacji). Może to być konsekwencją błędu programistycznego, ale w praktycznie implementowalnym modelu raczej wskazuje na niekompletną wiedzę systemu o wartościowaniu zmiennych.

W trakcie wykonywania programu zbiór zmiennych zwartościowanych na ogół poszerza się w miarę realizacji kolejnych operacji przepływu danych.

W ramach niniejszej pracy często stosować będziemy pojęcie *struktura danych* zamiennie z pojęciem zmiennej, a także pojęcia *wartość danej* i *dane* zamiennie z pojęciem *wartości*.

4.4 MASZYNA STEROWANA PRZEPŁYWEM DANYCH DLA POTRZEB ANIMACJI ALGORYTMÓW

Tradycyjne debuggery i narzędzia wizualizacji algorytmów pobierają z poddawanego analizie (obserwacji) oprogramowania dane w postaci sekwencji chwilowych wartości poszczególnych zmiennych. W proponowanej maszynie projekcyjnej, zaimplementowanej w systemie *Daphnis*, strumień danych wejściowych stanowi ciąg cząstkowych informacji o występujących w programie operacjach przepływu danych. Informacje te są zbierane w sposób dynamiczny, podczas wykonywania wizualizowanego programu.

4.4.1 UZUPEŁNIENIA TEKSTU ŹRÓDŁOWEGO

W systemie *Daphnis* animacja jest podtrzymywana przez serię kolejno wykonywanych wywołań podprogramów raportujących elementarne operacje przepływu danych. Przed pod-

daniem wizualizacji jakiejkolwiek zmiennej musi ona zostać najpierw zarejestrowana, tak by system mógł ją zidentyfikować i następnie śledzić jej wartości. Po zakończeniu wizualizacji zmienną należy wyrejestrować; powinno to nastąpić przed zakończeniem jej cyklu życiowego. Tak dochodzimy do trzech operacji, elementarnych dla naszego podejścia:

- **Register** – rozpoczyna wizualizację wskazanej zmiennej. Parametry operacji specyfikują adres (dokładniej: referencję) zmiennej poddawanej wizualizacji, w powiązaniu z jej nazwą symboliczną. Opcjonalny, trzeci parametr znajduje zastosowanie przy rejestracji zmiennych tablicowych, i określa wtedy rozmiar tablicy.
- **Play** – informuje system o wystąpieniu elementarnej operacji przepływu danych. Zmienne biorące udział w operacji są identyfikowane poprzez ich adresy i mogą być łatwo rozpoznane przy pomocy bazy danych zarejestrowanych zmiennych. Operacja pobiera co najmniej jeden parametr: adres zmiennej docelowej, oraz ewentualny zbiór adresów zmiennych źródłowych.
- **UnRegister** – informuje system, że zmienna nie powinna dłużej być poddawana wizualizacji. Funkcja ta powinna zostać wywołana tuż przed zwolnieniem zmiennej.

Wymienione trzy operacje stanowią kompletny zestaw niezbędnych funkcji, o których wywołania należy poszerzyć źródłowy tekst programu poddawanego wizualizacji. Zauważmy, że – w odróżnieniu od tradycyjnych narzędzi – funkcja *Play* informuje system o zachodzących przepływach danych, nie zaś o zmianach wartości.

Oto przykładowy fragment tekstu źródłowego programu realizującego sortowanie bąbelkowe (ang. *bubble sort*), poszerzony o adnotacje niezbędne dla systemu wizualizacji (fragmenty dodane są wyróżnione podkreśleniem):

```
const int N = 35;
int arr[N];                          Register("arr", arr[0], N);
int i, j;                            Register("i", i); Register("j", j);

... // inicjalizacja tablicy

for (i = 0; i < N - 1; i++)          { Play(i);
   for (j = N - 1; j > i; j--)       { Play(j);
         if (arr[j] < arr[j-1])
   {
              int tmp = arr[j];      Register("tmp", tmp);
                                     Play(&tmp, &arr[j]);
              arr[j] = arr[j-1];     Play(&arr[j], &arr[j-1]);
              arr[j-1] = tmp;        Play(&arr[j-1], &tmp);
   }                                 }}
                                     UnRegister(arr[0]);
```

4.4.2 SPECYFIKACJA WIZUALIZACJI

Oprócz uzupełnionego tekstu źródłowego programu, użytkownik – wizualizator powinien dostarczyć systemowi wizualizacji skrypt konfiguracyjny. W systemie *Daphnis* ma on formę pliku tekstowego. W ramach skryptu, dla każdej zmiennej przeznaczonej do wizualizacji specyfikuje się jej reprezentację graficzną oraz sposób translacji jej wybranych cech na określone parametry obrazu graficznego[3]. Zmienne w skrypcie konfiguracyjnym są identyfikowane poprzez nazwy symboliczne, które mogą, choć nie muszą, nawiązywać do ich prawdziwych identyfikatorów.

Procedury rejestracji zmiennych pozwalają systemowi na stworzenie bazy danych zawierającej aktualne informacje o zestawie zmiennych poddawanych wizualizacji w danej chwili. Symboliczne nazwy zmiennych, używane w dostarczonym przez użytkownika skrypcie konfiguracyjnym, są też obecne w wywołaniu funkcji *Register*. Dzięki temu możliwe jest powiązanie zarejestrowanej zmiennej z jej reprezentacją graficzną i z regułami translacji zdefiniowanymi w skrypcie. Wspomniana baza danych stanowi więc również i wewnętrzną reprezentację skryptu konfiguracyjnego. Dla każdej zmiennej biorącej w danej chwili udział w wizualizacji zawiera ona następujące informacje:

- nazwę symboliczną,
- adres,
- specyfikację reprezentacji graficznej i translacji cech zmiennej na parametry obrazu.

Ostatnia pozycja, stanowiąca reprezentację informacji pobranych ze skryptu konfiguracyjnego, jest przechowywana w postaci specjalnego obiektu. Jest on nie tylko odpowiedzialny za przechowanie informacji, ale także steruje procesem tworzenia i wyświetlania reprezentacji graficznej oraz translacji danych. Obiekt ten nazwaliśmy **aktorem**[4].

4.4.3 GENEROWANIE OBRAZU

Reprezentację graficzną zmiennych stanowią, najogólniej ujmując, elementy graficzne, widoczne na ekranie. Elementy te, w zależności od klasy, do której należą, charakteryzują się

---

[3] Produktem końcowym wizualizacji może być też inne, niż graficzne, przedstawienie algorytmu. Typowym przykładem jest udźwiękowienie (ang. auralisation), to jest przedstawienie za pomocą dźwięku (por. rozdział 1.2.2). Mimo, że system Daphnis oferuje tego typu „wizualizację dźwiękową", w tym miejscu, dla uproszczenia, ograniczymy się do reprezentacji graficznej.

[4] Nazwa „aktor" nie ma nic wspólnego z koncepcją stworzoną przez Atkinsona i Hewitta [164] i rozwiniętej dla potrzeb wizualizacji algorytmów przez Reynoldsa [130].

różnym kształtem, właściwościami, a także zbiorami atrybutów, takich jak położenie (współrzędna x i y), wysokość, szerokość, kolor, wyświetlany tekst itd. Zadaniem obiektu aktora jest dynamiczne przetworzenie cech zmiennej poddawanej wizualizacji na atrybuty jej reprezentacji graficznej.

Atrybut elementu graficznego może być powiązany z następującymi cechami zmiennej przez niego reprezentowanej:

- Wartość – może być stosunkowo łatwo, przy zastosowaniu metod translacji wyspecyfikowanych w skrypcie, przeliczona na określony atrybut. Na przykład, zmienna *zm* przyjmująca w programie wartości z zakresu *(a, b)* może być liniowo odwzorowana na wysokość elementu graficznego w zakresie (h_a, h_b).
- Adres (położenie) – zasadniczo odpowiada zawsze tej samej wartości określonego atrybutu. Na przykład, zmienna *zm* może być odwzorowana na atrybuty określające położenie elementu w miejscu o współrzędnych (x_{zm}, y_{zm}). Wartość atrybutu może być wyspecyfikowana *a priori*, lub związana, na przykład z indeksem zmiennej składowej tablicy.
- Trzeci przypadek stanowią atrybuty nie powiązane, ustawione *a priori*.

Podczas wizualizacji sterowanej przepływem danych, obiekt aktora jest raczej związany z wartością przechowywaną w zmiennej, a nie z tą zmienną. Podczas wizualizacji operacji przepływu danych nie tylko wartość może ulec zmianie, ale również i jej adres – mówiąc obrazowo, wraz z wartością przepływa również jej aktor. Mamy więc do czynienia z następującymi sytuacjami:

- Zmiana wartości (np. $a = 10$) – wiąże się z taką transformacją reprezentacji graficznej kontrolowanej przez aktora, by w jej wyniku atrybuty graficzne związane z wartością odpowiadały nowej wartości zmiennej. Na ogół wyraża się to deformacją obiektu graficznego, na przykład zmianą jego wysokości.
- Zmiana adresu w wyniku operacji przepływu danych (np. $a = b$) – wiąże się z taką transformacją reprezentacji graficznej zmiennej źródłowej kontrolowanej przez odpowiedniego aktora, by w jej wyniku atrybuty graficzne związane z adresem odpowiadały nowemu adresowi. Nową wartość tych atrybutów określa się odnajdując aktora związanego z wartością dotychczas przechowywaną w tej zmiennej. Element graficzny reprezentujący zmienną źródłową przejmuje więc atrybuty adresu od elementu reprezentującego zmienną docelową. Ponieważ są to najczęściej atrybuty określające współrzędne obiektu na ekranie, operacja przepływu danych jest na ogół obserwowana jako przemieszczenie obiektu reprezentującego zmienną źró-

dłową w kierunku zmiennej docelowej (często z równoczesną deformacją odzwierciedlającą zmianę wartości). Możliwe jest też użycie innych atrybutów, co jest wykorzystywane w niektórych bardziej wymyślnych, wysoce analitycznych wizualizacjach.

- Kombinacja obydwu powyższych sytuacji.

Dokładniejszy opis mechanizmu generowania obrazu graficznego w systemie *Daphnis* można znaleźć w rozdziale 6.4, a szczegółowy opis implementacji aktorów i innych składników systemu – w rozdziale 7.2.

4.4.4 GRAFICZNA INTERPRETACJA OPERACJI ELEMENTARNYCH

Po każdej odnotowanej operacji przepływu danych, baza zarejestrowanych zmiennych jest wykorzystywana do odnalezienia – na podstawie adresów zmiennych biorących udział w operacji przepływu – obiektów aktorów odpowiedzialnych za wygenerowanie graficznych reprezentacji tych zmiennych. Stosowany jest następujący algorytm:

- Jeśli zmienna docelowa jest wartościowana przez wartość *a priori* (np. $a = 5$), jej reprezentacja graficzna jest modyfikowana przez aktora tak, by odzwierciedlić nową wartość zmiennej. Nie jest przedstawiany żaden przepływ danych – sytuacja jest bardzo podobna do spotykanej w przypadku tradycyjnych debuggerów oraz prostej wizualizacji.
- Jeśli zmienna docelowa jest modyfikowana wyłącznie na podstawie poprzedniej swojej wartości (tzn. jest jednocześnie jedyną zmienną źródłową), stosowana jest procedura jak podana wyżej.
- Jeśli jest wyspecyfikowana co najmniej jedna zmienna źródłowa, i jest ona zwartościowana i różna od zmiennej docelowej, wówczas zachodzi przypadek wizualizacji właściwego przepływu danych. Obiekt reprezentujący zmienną lub zmienne źródłowe jest powielany i płynnie przesuwany w kierunku obiektu reprezentującego zmienną docelową, przy czym równocześnie jest poddawany niezbędnym modyfikacjom, oddającym ewentualne zmiany wartości danych. Poprzednia reprezentacja zmiennej docelowej (jeśli istniała) zanika do czasu ukończenia animowanego ruchu. Zmienna docelowa przejmuje reprezentację graficzną i zasady translacji zmiennej źródłowej (lub pierwszej z nich, jeśli było ich więcej niż jedna), utrzymuje natomiast swoją pozycję na ekranie.

Zarejestrowane, lecz nie zwartościowane zmienne nie posiadają reprezentacji graficznej. Jeśli są one wartościowane *a priori*, nowa reprezentacja graficzna jest tworzona dla nich od razu. Jeśli stanowią one miejsce przeznaczenia przepływu danych, stosowana jest procedura

jak podano wyżej, z tym, że nie ma oczywiście potrzeby usuwania poprzedniej reprezentacji. Jeśli źródłem przepływu danych są zmienne niezwartościowane, są one po prostu ignorowane.

4.5 ZASTOSOWANIE SIECI PETRIEGO DO OPISU DZIAŁANIA ALGORYTMÓW

Zaproponowany w poprzednich punktach algorytm wizualizacji algorytmów zapewnia spełnienie postulatu wizualizacji ilustrującej przepływy danych. Aby spełnić dwa pozostałe wymagania wprowadzone w rozdziale 4.2, to jest przestrzenną oraz temporalną redukcję informacji, musieliśmy sięgnąć do formalizmu sieci Petriego, przy jego zastosowaniu opisać proces wykonania algorytmu, i dopiero to pozwoliło nam ostatecznie sformułować algorytm wizualizacji.

Sieci Petriego są algebraiczną i graficzną strukturą zaproponowaną przez Carla Adama Petriego w 1962 roku [149]. Zostały one opisane w licznych publikacjach [150 – 153]. Sieci Petriego są doskonałym środkiem opisu systemów sterowanych przepływem danych. Systemy sterowane przepływem operacji, ze swoją ściśle zdeterminowaną kolejnością wykonywania operacji i stosunkowo niskim poziomem zrównoleglenia, nie stanowią dobrego przedmiotu dla opisu przy pomocy sieci Petriego. Jednak naszym celem jest zbudowanie narzędzia pozwalającego między innymi na temporalną redukcję informacji, a zatem na usunięcie ścisłego zdeterminowania kolejności operacji i, co za tym idzie, istotne zrównoleglenie realizacji poddawanego wizualizacji algorytmu. Do tego celu sieci Petriego okazały się bardzo pożytecznym narzędziem projektowym.

Sieci Petriego były już wykorzystywane do celów związanych z algorytmiką. Obszerny raport z 15-letnich doświadczeń w zastosowaniu ich zagadnieniach związanych z szeroko pojętym projektowaniem można znaleźć w pracy R.M. Shapiro [162]. Wczesne próby wykorzystania sieci Petriego dotyczyły tłumaczenia programów translatora Fortranu na sieci [163] oraz stworzenia modelu sieci Petriego dla systemu operacyjnego SCOPE 3.2 [159]. Bardzo ciekawe były prace G. Roucairola na temat zrównoleglenia i redukowania programów [160, 161]. Z dziedzin pokrewnych wymienić należy zastosowanie sieci Petriego do opisu i analizy semantyki języków programowania [154, 156, 158] oraz do analizy i projektowania cyfrowych układów przełączających [155].

4.5.1 DEFINICJE

Chociaż formalizm sieci Petriego jest powszechnie znany, dla ścisłości przytoczymy jednak podstawowe definicje.

DEFINICJA 4.1

Trójkę *PN (P, T, F)* nazywamy **siecią Petriego** wtedy i tylko wtedy, gdy:

- P i T są zbiorami rozłącznymi. Elementy zbioru P nazywamy **miejscami**, elementy zbioru T nazywamy **tranzycjami**.
- $F \subseteq (P \times T) \cup (T \times P)$ jest relacją dwuargumentową, zwaną **relacją przepływu sieci**.

Miejsca reprezentujemy graficznie kółkami, tranzycje zaś – prostokątami. Relację przepływu reprezentuje się łukami łączącymi odpowiednie kółka i prostokąty.

DEFINICJA 4.2

Niech *PN (P, T, F)* będzie siecią, oraz $x, y \in P \cup T$.

- $^*x = \{ y \mid \ y F x \}$ nazywamy **zbiorem wejściowym** x.
- $x^* = \{ y \mid \ x F y \}$ nazywamy **zbiorem wyjściowym** x.

DEFINICJA 4.3

Dla danej sieci *PN (P, T, F)* **znakowaniem** nazywamy funkcję $\Omega: P \to N \cup \{0\}$, która każdemu miejscu sieci przyporządkowuje liczbę całkowitą nieujemną.

W dalszym ciągu rozpatrywać będziemy sieci, dla których zbiorem wartości funkcji znakowania jest zbiór $\{0, 1\}$. Miejsca, dla których funkcja ta przyjmuje wartość *1*, oznaczamy graficznie poprzez umieszczenie kropki wewnątrz reprezentującego je kółka i mówimy, że zawierają one **żeton**.

DEFINICJA 4.4

Niech *PN (P, T, F)* będzie siecią. Tranzycję $t \in T$ nazywamy **aktywną** wtedy i tylko wtedy, gdy znakowanie wszystkich miejsc należących do zbioru wejściowego tej tranzycji jest niezerowe (innymi słowy: wszystkie wejściowe miejsca posiadają żeton):

$$\forall p \in {}^*t: \quad \Omega(p) > 0$$

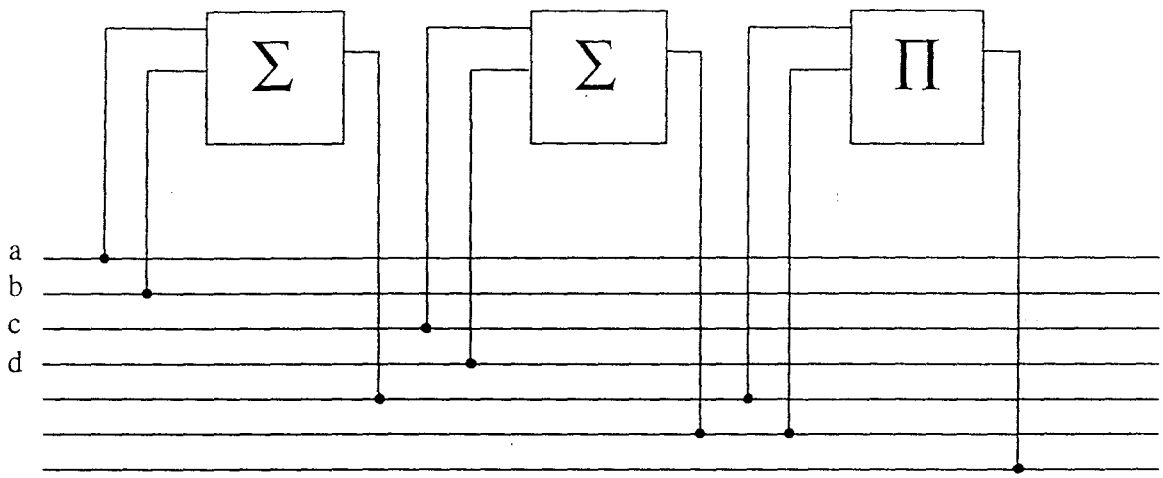
DEFINICJA 4.5

Niech *PN (P, T, F)* będzie siecią, a $t \in T$ aktywną tranzycją w tej sieci. **Odpaleniem tranzycji *t*** nazywamy zdarzenie, w wyniku którego wszystkie miejsca wejściowe tej tranzycji tracą po jednym żetonie, a miejsca wyjściowe jeden żeton otrzymują:

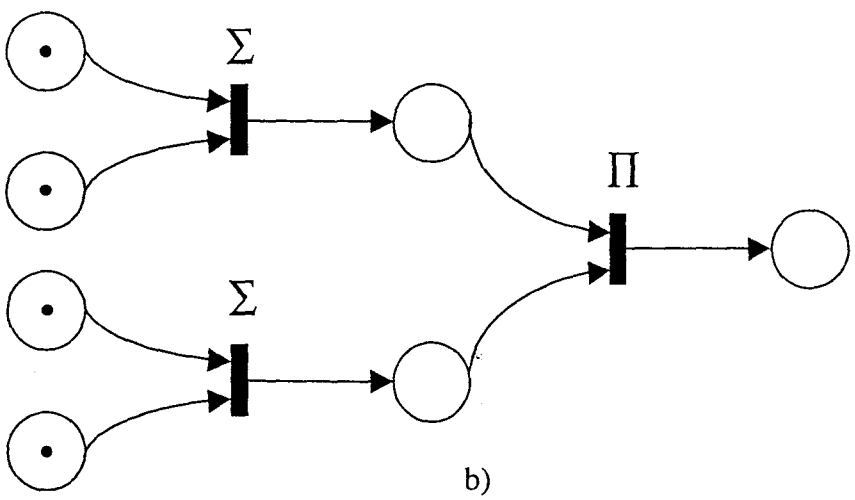
$$\forall p \in {}^*t: \quad \Omega_1(p) = \Omega(p) - 1$$
$$\forall p \in t^*: \quad \Omega_1(p) = \Omega(p) + 1$$

4.5.2 OPIS SYSTEMÓW STEROWANYCH PRZEPŁYWEM DANYCH

Teoretyczny model maszyny sterowanej przepływem danych [182, 183] przedstawia sobą wysoce zrównoleglona strukturę złożoną z wielu procesorów (rys. 4.1). Każdy procesor posiada pewną liczbę wejściowych linii danych, reprezentujących argumenty, i wyjściowych li-



a)



b)

Rys. 4.1. System sterowany przepływem danych, obliczający wartość wyrażenia *(a+b)*(c+d)*: a) schemat; b) reprezentacja w postaci sieci Petriego

nii danych, reprezentujących wyniki, i jest przeznaczony do obliczenia wyników na podstawie argumentów. Wyniki obliczane przez określone procesory stanowią dane wejściowe dla innych procesorów. W stanie początkowym wszystkie wejściowe linie danych reprezentujące argumenty dla całego algorytmu uznawane są za zwartościowane – wszystkie pozostałe nie są zwartościowane. Podczas pracy, każdy z procesorów oczekuje na skompletowanie wszystkich argumentów, czyli zwartościowanie wszystkich linii wejściowych; następnie wykonuje obliczenie na podstawie argumentów i odpowiednio wartościuje swoje linie wyjściowe. Wszystkie procesory działają współbieżnie. W stanie początkowym przynajmniej jeden procesor musi być zdolny do obliczenia wyników, to znaczy, jego zbiór argumentów musi być podzbiorem zbioru argumentów początkowych całego algorytmu; w przeciwnym przypadku proces obliczeniowy nie będzie się mógł rozpocząć. Podczas obliczeń, wyniki dostarczane przez kolejne procesory dostarczają argumentów innym procesorom; taka propagacja obliczeń utrzymuje system w ruchu aż do momentu wyprodukowania wyników uważanych za końcowe rezultaty działania algorytmu.

Zasada sterowania przepływem danych znalazła wiele praktycznych zastosowań. Pomiędzy nimi do najważniejszych należy zaliczyć środowiska i języki programowania wizualnego [40, 43], w których programy są budowane z bloków działających w sposób analogiczny do opisanych wyżej procesorów. Przykładem takiego środowiska jest także system *Siamoa*, dokładniej opisany w rozdziale 3.3. Innym bardzo interesującym zastosowaniem są superkomputery oparte na architekturach sterowanych przepływem danych. Dwie najlepiej znane realizacje to architektura zaproponowana przez J. Dennisa i jego grupę w MIT [176, 178, 179], oraz tzw. maszyna z Manchester, zaprojektowana i zrealizowana przez J.R. Gurda z Uniwersytetu w Manchester [177, 179]. Ich komputery zawierają zbiorniki operacji oczekujących na skompletowanie argumentów, zbiorniki argumentów i zestaw pracujących współbieżnie procesorów. Układ sterujący dobiera argumenty do oczekujących operacji i po ich skompletowaniu przekazuje operacje wraz z argumentami wolnemu procesorowi do wykonania; ten, po wykonaniu obliczeń, wprowadza rezultaty z powrotem do zbiornika argumentów, gdzie stają się dostępne dla kolejnych operacji. Podobna architektura została zastosowana do organizacji rozproszonych obliczeń w heterogenicznej sieci komputerowej [180, 181].

Struktura sterowana przepływem danych może być reprezentowana za pomocą sieci Petriego (rys. 4.1), w której:

- Miejsca reprezentują linie danych.
- Tranzycje reprezentują procesory.
- Zbiory wejściowe tranzycji odpowiadają zbiorom linii wejściowych danych procesorów.
- Zbiory wyjściowe tranzycji odpowiadają zbiorom linii wyjściowych danych procesorów.
- Znakowanie sieci odzwierciedla zwartościowanie linii danych. Miejsca zawierające żeton reprezentują te linie danych, które są zwartościowane.

Początkowe znakowanie sieci odzwierciedla początkowy stan systemu: żetony są umieszczone w tych miejscach, które reprezentują dane wejściowe dla całego algorytmu. Odpalenie tranzycji odpowiada przeprowadzeniu obliczenia przez procesor. Wszystkie miejsca należące do zbioru wejściowego tranzycji muszą posiadać żeton, podobnie jak wszystkie linie wejściowe muszą być zwartościowane. W wyniku odpalenia tranzycji wszystkie miejsca należące do zbioru wyjściowego tranzycji otrzymują żeton, podobnie jak w wyniku wykonania obliczenia wartościowane są wszystkie linie wyjściowe procesora.

Model oparty o sieć Petriego oddaje dwie istotne cechy systemów sterowanych przepływem danych: równoległość wykonywania procesu obliczeniowego i niedeterminizm kolejności wykonywania obliczeń.

4.5.3 OPIS SYSTEMÓW STEROWANYCH PRZEPŁYWEM OPERACJI

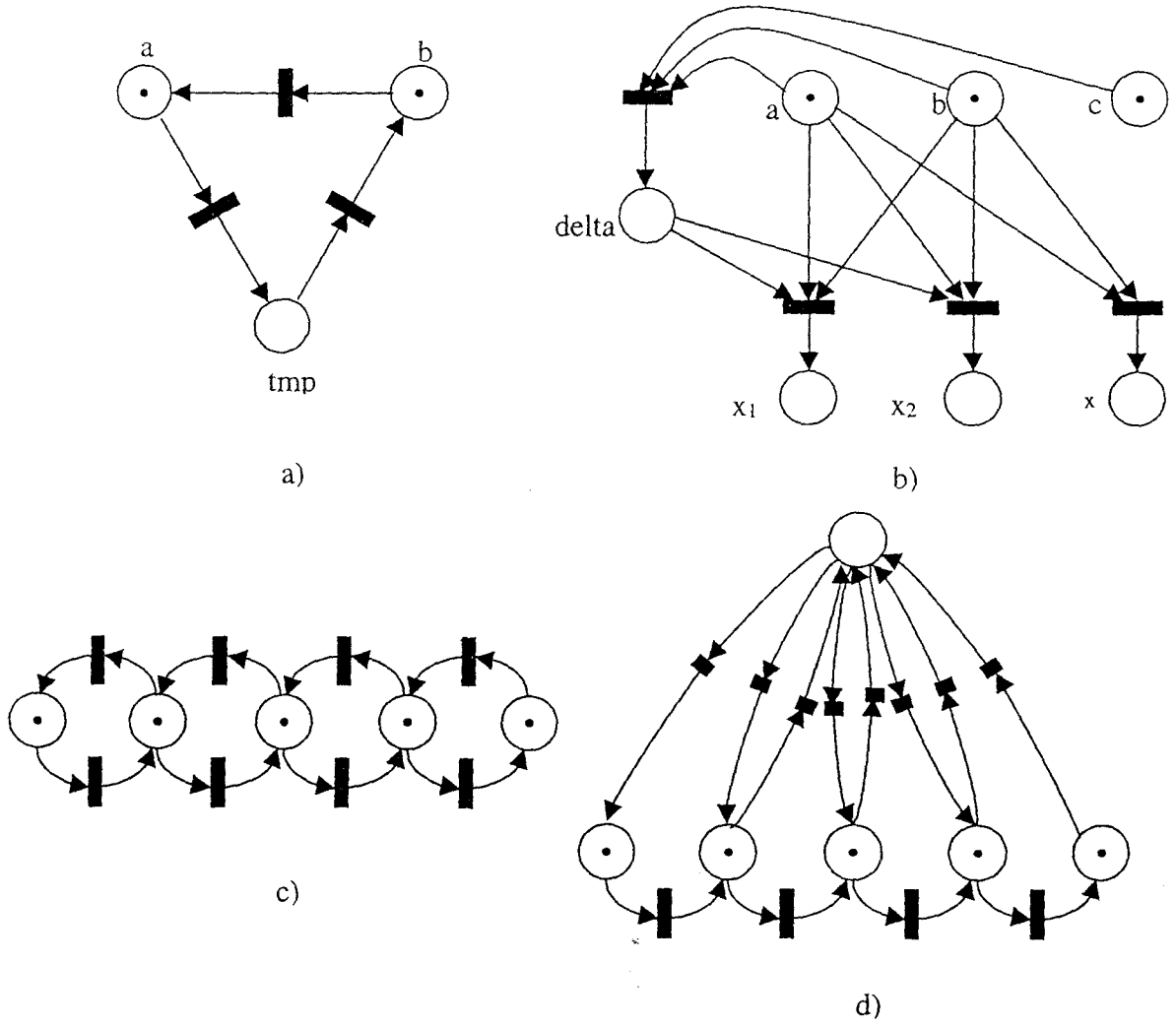
Tradycyjne komputery są sterowane przepływem operacji, a nie danych. Kolejność wykonywania obliczeń jest ściśle zdeterminowana, a poziom równoległości jest stosunkowo niski: biorąc pod uwagę krótsze sekwencje operacji, w większości przypadków w ogóle nie spotykamy ich zrównoleglenia.

Utwórzmy model przepływu danych zachodzących w procesie wykonania algorytmu w systemie sterowanym przepływem operacji, posługując się siecią Petriego $PN = (P, T, F)$. Niech zbiór P miejsc odpowiada zbiorowi zmiennych biorących udział w programie, a zbiór T tranzycji – zbiorowi wszystkich operacji przepływu danych zachodzących w procesie wykonania algorytmu. Wówczas relacja przepływu sieci F jest zdefiniowana w następujący sposób:

- Dla każdego $p_i \in P$, $t_j \in T$ relacja przepływu $p_i \; F \; t_j$ jest spełniona wtedy i tylko wtedy, gdy zmienna reprezentowana przez p_i jest zmienną źródłową w operacji przepływu danych reprezentowanej przez t_j.

- Dla każdego $t_j \in T$, $p_k \in P$ relacja przepływu $t_j F p_k$ jest spełniona wtedy i tylko wtedy, gdy zmienna reprezentowana przez p_k jest zmienną docelową w operacji przepływu danych reprezentowanej przez t_j.

Dla każdej tranzycji reprezentującej operację przepływu danych t_j, zbiór zmiennych źródłowych p_i odpowiada zbiorowi wejściowemu tranzycji $*t_j$, a zmienna docelowa p_k stanowi jednoelementowy zbiór wyjściowy tranzycji t_j*. W stanie początkowym, umieszczamy żetony we wszystkich miejscach reprezentujących zmienne zwartościowane w stanie początkowym, to jest przechowujące dane wejściowe całego algorytmu. Przykładowe sieci Petriego reprezentujące kilka algorytmów przedstawiamy na rys. 4.2.



Rys. 4.2. Przykłady reprezentacji przepływów danych w algorytmach za pomocą sieci Petriego: a) operacja zamiany; b) równanie kwadratowe; c) uproszczony widok sortowania bąbelkowego; d) sortowanie bąbelkowe ze zmienną pomocniczą

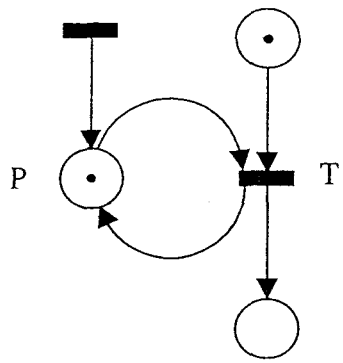
Odpalenie tranzycji w tradycyjnej sieci Petriego powoduje, że wszystkie miejsca wejściowe tej tranzycji tracą żeton. W przypadku naszego modelu oznaczałoby to, że po wystąpieniu przepływu danych jego zmienne źródłowe przestają być zwartościowane. Aby uczynić nasz model bardziej zgodnym z rzeczywistością, wprowadziliśmy niewielką modyfikację: dla danej sieci *PN(P, T, F)* rozpatrujemy zamiast niej zmodyfikowaną sieć *PN'(P, T, F')*, przy czym relacja *F'* jest zdefiniowana następująco:

$$\forall(p \in P, t \in T): \quad pF't \Leftrightarrow pFt \ \wedge \ tF'p \Leftrightarrow tFp \vee pFt$$

Relacja *F'* spełnia więc następujący warunek:

$$\forall(p \in P,\ t \in T): \quad pF't \Rightarrow tF'p$$

W związku z tą zależnością, odpalenie dowolnej tranzycji powoduje wprawdzie odczyt wszystkich miejsc wejściowych, ale nie zmienia znakowania tych miejsc (por. rys. 4.3). Dla uproszczenia jednak będziemy się posługiwać nadal pierwotną siecią *PN*, mając na uwadze możliwą jej modyfikację.



Rys. 4.3. Odczyt miejsca bez zmiany znakowania

Jak wspomniano, sekwencja operacji przepływu danych jest ściśle zdeterminowana. Zatem również i odpalenia tranzycji muszą zachodzić w ściśle określonym porządku. Jednak odpalenie tranzycji w typowych sieciach Petriego nie jest czasowo zdeterminowane. Model oparty o taką sieć nie pozwala na określenie kolejności wykonywania operacji. Co więcej, w sieci każda aktywna (odblokowana) tranzycja może być odpalona. Wykonanie operacji w algorytmie jest natomiast zdeterminowane dodatkowymi, zdefiniowanymi w tym algorytmie, warunkami. Dla przykładu, w algorytmie sortowania możliwy jest niemal każdy przesył danych pomiędzy dowolnymi dwoma elementami sortowanego ciągu. Jednak aby skutecznie posortować ciąg, realizowany jest tylko określony podzbiór wszystkich możliwych operacji przepływu, i to w określonym porządku. Aby nasz model stał się w pełni zgodny z rzeczywistym przebiegiem procesu wykonania algorytmu, musimy się posłużyć **nieautonomicznymi**, lub **czasowymi sieciami Petriego** [150, 151]. Sieci takie opisują systemy, których ewolucja jest warunkowana zewnętrznymi zdarzeniami zachodzącymi w czasie. Odpalenie tranzycji jest możliwe tylko pod warunkiem wystąpienia określonego zdarzenia. W naszym modelu takim zdarzeniem jest rzeczywiste wykonanie operacji przepływu. Faktycznie, nieautonomiczne sieci Petriego są

niezbędne przy opisie całego algorytmu; w następnym podrozdziale użyjemy jednak zwykłej, to jest autonomicznej sieci do opisu odpowiednio dobranych wycinków algorytmu.

4.6 KROK ALGORYTMU I JEGO WŁAŚCIWOŚCI

Obecnie stworzymy podstawy do opisu zachowania systemu sterowanego przepływem operacji – za pomocą zwykłej, autonomicznej sieci Petriego. Ze względu na deterministyczne zależności czasowe charakterystyczne dla tej klasy systemów, nie możemy modelować całego procesu wykonania algorytmu, zatem ograniczymy się do tzw. **kroku algorytmu**, który jest pojęciem kluczowym dla redukcji informacji temporalnej. Pod koniec podrozdziału wprowadzimy też transformacje będące podstawą redukcji informacji przestrzennej.

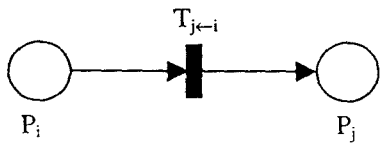
Najpierw jednak wprowadzimy stosowaną dalej notację.

NOTACJA 4.1

Niech $F_{j\leftarrow i}$ będzie operacją przepływu danych ze zmiennej źródłowej i do zmiennej docelowej j. Wówczas sieć *PN(P, T, F)* taką, że:

- $P = \{p_i, p_j\}$, miejsce p_i reprezentuje zmienną i, miejsce p_j reprezentuje zmienną j,
- $T = \{t_{j\leftarrow i}\}$, tranzycja $t_{j\leftarrow i}$ reprezentuje operację przepływu danych od i do j,
- $F = \{(p_i, t_{j\leftarrow i}), (t_{j\leftarrow i}, p_j)\}$,

oznaczamy krótko jako $\{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$ (rys. 4.4).



Rys. 4.4. Sieć $\{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$

NOTACJA 4.2

Dla danej sieci *PN(P, T, F)* zapisujemy, że:

$$PN' = PN \cup \{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$$

wtedy i tylko wtedy, gdy *PN'(P', T', F')* jest taką siecią, że:

- $P' = P \cup \{p_i, p_j\}$
- $T' = T \cup \{t_{j\leftarrow i}\}$
- $F' = F \cup \{(p_i, t_{j\leftarrow i}), (t_{j\leftarrow i}, p_j)\}$

Mówimy wówczas, że $\{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$ jest **podsiecią** sieci *PN'*, co zapisujemy jako:

$$\{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\} \in PN'$$

NOTACJA 4.3

Dla danej sieci *PN(P, T, F)* zapisujemy, że:

$$PN' = PN - \{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$$

wtedy i tylko wtedy, gdy

$$PN = PN' \cup \{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}.$$

4.6.1 KROK ALGORYTMU

Załóżmy, że algorytm jest wykonywany na maszynie wieloprocesorowej w taki sposób, że kilka kolejnych operacji tego algorytmu może być wykonywanych współbieżnie. Ze względu na semantykę algorytmu niektóre operacje trzeba jednak wykonywać po zakończeniu innych operacji. Tak więc proces wykonania algorytmu dzieli się na sekwencję etapów, w trakcie których kolejne podzbiory operacji są wykonywane w sposób współbieżny. Każdy z takich podzbiorów nazywamy **krokiem algorytmu**. Pojęcie to stosować będziemy też do sekwencyjnej realizacji algorytmu: fakt przynależność operacji do tego samego kroku mówi nam wówczas o potencjalnej możliwości wykonania tych operacji w sposób współbieżny.

DEFINICJA 4.6

Krokiem algorytmu nazywamy taki zbiór kolejno wykonywanych operacji, że możliwa jest zmiana realizacji algorytmu polegająca na wykonaniu ich w dowolnej, niezdeterminowanej kolejności, albo też współbieżnie (bez zmiany semantyki algorytmu).

PRZYKŁAD 4.4

Rozważmy następujący fragment programu (zapisanego w języku C):

```
1: a = 5;
2: b = x;
3: c = x;
4: c = 10;
5: z = c;
```

Łatwo zauważyć, że instrukcje (*1*), (*2*) i (*3*) mogą być wykonane w dowolnej kolejności lub też współbieżnie. Instrukcja (*4*) musi jednak następować po instrukcji (*3*), instrukcja (*5*) zaś po instrukcji (*4*). Powyższa sekwencja składa się z trzech kroków, przy czym pierwszy krok obejmuje instrukcje (*1*), (*2*) i (*3*), drugi – instrukcję (*4*), a ostatni – instrukcję (*5*).

Aby określić zasady podziału sekwencji operacji algorytmu na kroki, skorzystamy z modelu korzystającego z sieci Petriego, który został opisany w poprzednim punkcie.

LEMAT 4.1

Niech *PN (P, T, F)* będzie siecią Petriego modelującą pewien ciąg operacji kolejno wykonywanych przez system sterowany przepływem operacji. Podzbiór operacji odwzorowany przez sieć *PN* stanowi krok algorytmu wtedy i tylko wtedy, gdy:

$$\forall t_1, t_2 \in T: \quad t_1 \neq t_2 \quad \Rightarrow \quad t_1^* \cap t_2^* = t_1^* \cap {}^*t_2 = \varnothing$$

gdzie:

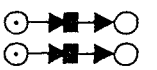
$^*t$ – zbiór wejściowy tranzycji t

t^* – zbiór wyjściowy tranzycji t

DOWÓD:

Niech $t_1, t_2 \in T \wedge t_1 \neq t_2$. Pokażemy, że warunek $t_1^* \cap t_2^* = t_1^* \cap {}^*t_2 = \varnothing$ jest konieczny i wystarczający na to, by kolejność wykonania operacji reprezentowanych przez t_1 i t_2 nie miała wpływu na semantykę algorytmu. W tym celu rozpatrzymy następujące przypadki szczególne:

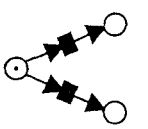
1. Tranzycje odseparowane:

   $t_1^* \cap t_2^* = t_1^* \cap {}^*t_2 = {}^*t_1 \cap {}^*t_2 = \varnothing$

   W tym przypadku obydwie operacje działają na zupełnie różnych zbiorach zmiennych, są od siebie całkowicie niezależne (np. $a = b$; $c = d$). Kolejność ich wykonania nie jest istotna, mogą też być wykonywane równocześnie.

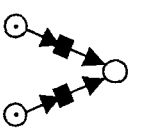
2. Tranzycje w konflikcie:

   $t_1^* \cap t_2^* = t_1^* \cap {}^*t_2 = \varnothing \quad \wedge \quad {}^*t_1 \cap {}^*t_2 \neq \varnothing$

   Zgodnie z propozycją poszerzenia relacji przepływu F można założyć niedeterministyczne odpalenie obu tranzycji. Z punktu widzenia semantyki algorytmu, odpowiada to dwukrotnemu odczytowi tej samej zmiennej (np. $a = x$; $b = x$): są to operacje niezależne, mogą być wywoływane w dowolnej kolejności a także równocześnie.

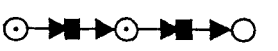
3. Tranzycje w konflikcie przed sytuacją kontaktową:

   $t_1^* \cap t_2^* \neq \varnothing$

   Tego rodzaju sytuację kontaktową interpretujemy jako dwa kolejne zapisy do tej samej zmiennej (np. $x = a$; $x = b$). Efekt końcowy, to jest wartość zmiennej docelowej, zależy od kolejności wykonania obu operacji.

4. Tranzycje w sytuacji kontaktowej:

   $t_1^* \cap {}^*t_2 \neq \varnothing$

Sytuacja kontaktowa może być różnie interpretowana w zależności od kolejności odpalenia tranzycji (np. jako $x=a;\ b=x$ lub $b=x;\ x=a$).

Za pomocą przypadków (1) i (2) udowodniliśmy, że wzmiankowany wyżej warunek jest wystarczający. Za pomocą przypadków (3) i (4) wykazaliśmy, że jest również konieczny.

4.6.2 KROK SYNCHRONICZNY ALGORYTMU

Jeśli założymy, że wszystkie operacje należące do kroku algorytmu są wykonywane równolegle i synchronicznie, możemy wówczas, nie zmieniając semantyki algorytmu, dołączyć do tego kroku dodatkowe operacje. Wynika to z faktu, że są operacje, które nie mogłyby być wykonane współbieżnie, asynchronicznie (zatem nie należą do tego samego kroku algorytmu), ale mogą być wykonywane w trybie współbieżnym, synchronicznym. Tak rozszerzony krok algorytmu określamy mianem **kroku synchronicznego**.

DEFINICJA 4.7

Krokiem synchronicznym algorytmu nazywamy taki ciąg kolejno wykonywanych operacji, że możliwa jest zmiana realizacji algorytmu polegająca na wykonaniu ich równolegle, synchronicznie (bez zmiany semantyki algorytmu).

PRZYKŁAD 4.5

Rozważmy następujący fragment programu (zapisanego w języku C):

```
1: aux = a;
2: a = b;
3: b = aux;
```

Łatwo zauważyć, że żadne dwie z powyższych instrukcji nie należą do tego samego zwykłego kroku algorytmu. Jednak instrukcje (*1*) i (*2*) mogą być wykonane synchronicznie, podobnie zresztą jak instrukcje (*2*) i (*3*). Jedynie instrukcje (*1*) i (*3*) muszą być wykonane jedna po drugiej. Tak więc możliwe podziały sekwencji na kroki synchroniczne to: (*1-2*) i (*3*) lub (*1*) i (*2-3*).

LEMAT 4.2

Niech *PN (P, T, F)* będzie siecią Petriego modelującą pewien ciąg operacji kolejno wykonywanych przez system sterowany przepływem operacji, a funkcja $\tau(t)$ określa czas odpalenia tranzycji t. Podzbiór operacji odwzorowany przez sieć *PN* stanowi krok synchroniczny algorytmu wtedy, i tylko wtedy, gdy:

$$\forall t_1, t_2 \in T:\ \tau(t_1) < \tau(t_2) \ \Rightarrow\ t_1^* \cap t_2^* = t_1^* \cap {}^* t_2 = \varnothing$$

DOWÓD:

Nie będziemy przytaczać całego dowodu, gdyż jest on analogiczny do przytoczonego poprzednio dowodu dotyczącego zwykłego kroku algorytmu. Istotniejsza różnica pojawia się tylko dla sytuacji kontaktowej, kiedy dla przypadku:

$$t_1, t_2 \in T \quad \wedge \quad t_1 \neq t_2 \quad \wedge \quad t_1{}^* \cap {}^* t_2 \neq \varnothing$$

obydwie operacje pozostają w obrębie tego samego kroku synchronicznego, jeśli nie spełniona jest zależność $\tau(t_1) < \tau(t_2)$. Odpowiada to sekwencji instrukcji typu: *b=x; x=a*. Kolejność wykonania tych instrukcji nie może być odwrócona (zmieniłoby to końcową wartość *b*), dopuszczalne jest jednak wykonanie równoległe, synchroniczne.

Na zakończenie zdefiniujemy jeszcze dwa pojęcia i podamy lemat, w oparciu o który łatwiej będzie sformułować algorytm temporalnej redukcji informacji.

DEFINICJA 4.8

Niech *PN (P, T, F)* będzie siecią Petriego modelującą pewien ciąg operacji kolejno wykonywanych przez system sterowany przepływem operacji, a funkcja $\tau(t)$ określa czas odpalenia tranzycji *t*.

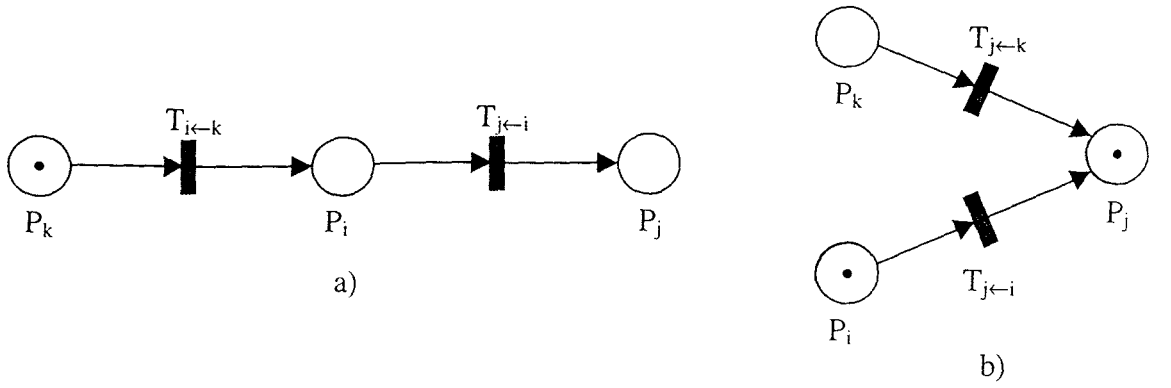
1. Operacje reprezentowane przez tranzycje $t_1, t_2 \in T$, przy czym $\tau(t_1) < \tau(t_2)$, nazywamy **zależnymi**, jeżeli $t_1{}^* \cap {}^* t_2 \neq \varnothing$. Mówimy, że zachodzi między nimi relacja **zależności** (rys. 4.5a).
2. Operacje reprezentowane przez tranzycje $t_1, t_2 \in T$ nazywamy **sekwencyjnymi**, jeżeli $t_1{}^* \cap t_2{}^* \neq \varnothing$. Mówimy, że zachodzi między nimi relacja **sekwencji** (rys. 4.5b)[5].

LEMAT 4.3

Operacje należą do tego samego kroku synchronicznego wtedy, i tylko wtedy, gdy nie są zależne ani sekwencyjne.

W przypadku operacji *zależnych* (np. *x = a; b = x*), działanie operacji wykonanej w drugiej kolejności jest uzależnione od wartości obliczonej przez pierwszą operację. W przypadku operacji sekwencyjnych (np. *x = a; x = b*), obydwie operacje wpisują kolejno różne wartości do tej samej zmiennej. Lemat 4.3 wynika wprost z lematu 4.2 i nie będziemy go udowadniać.

[5] W teorii szeregowania mówimy niekiedy o zależności przepływu (ang. *flow dependence*) oraz o zależności wyjść (ang. *output dependency*), które to terminy odpowiadają naszej *zależności* i *sekwencji*.



Rys. 4.5. Sytuacje, w których nowa tranzycja $T_{j\leftarrow i}$ nie może być odpalona równolegle: a) zależność; b) sekwencja

DEFINICJA 4.9

Siecią jednokrokową synchroniczną nazywamy sieć Petriego reprezentującą ciąg operacji należących do tego samego kroku synchronicznego algorytmu.

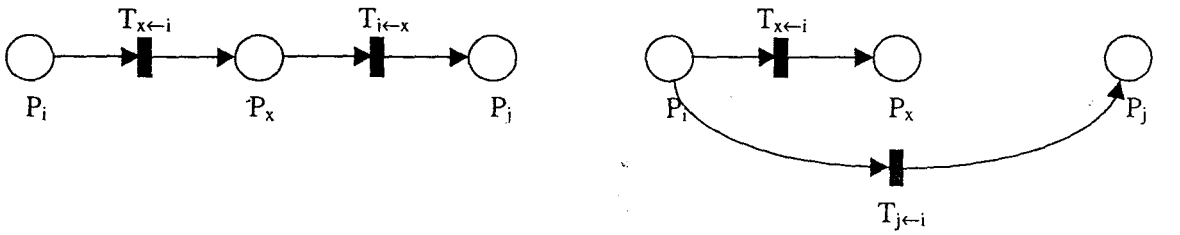
4.6.3 PRZESTRZENNE TRANSFORMACJE KROKU SYNCHRONICZNEGO

Zdefiniujemy dwa typy transformacji sieci reprezentującej ciąg operacji i podamy pewne ich właściwości, przydatne podczas przestrzennej redukcji informacji.

DEFINICJA 4.10

Niech *PN(P, N, F)* będzie siecią i niech $\{p_i \rightarrow t_{x\leftarrow i} \rightarrow p_x\} \cup \{p_x \rightarrow t_{j\leftarrow x} \rightarrow p_j\} \in PN$ (rys. 4.6, z lewej). **Sklejeniem $P_i - P_j$ wokół P_x** nazywamy taką transformację sieci *PN*, w wyniku której powstaje sieć *PN'*, w której operację $\{p_x \rightarrow t_{j\leftarrow x} \rightarrow p_j\}$ zastąpiono operacją $\{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$ (rys. 4.6, z prawej):

$$PN' = PN - \{p_x \rightarrow t_{j\leftarrow x} \rightarrow p_j\} \cup \{p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j\}$$



Rys. 4.6. Transformacja sklejenia $P_i - P_j$ wokół P_x

PRZYKŁAD 4.6

Sklejenie $P_i - P_j$ wokół P_x odpowiada zastąpieniu następującego fragmentu programu:

```
aux = i;
j = aux;
```

fragmentem:

```
j = i;
```

Zauważmy przy tym, że w równoległym i synchronicznym trybie wykonania algorytmu użycie zmiennej pomocniczej jest częstokroć nadmiarowe.

DEFINICJA 4.11

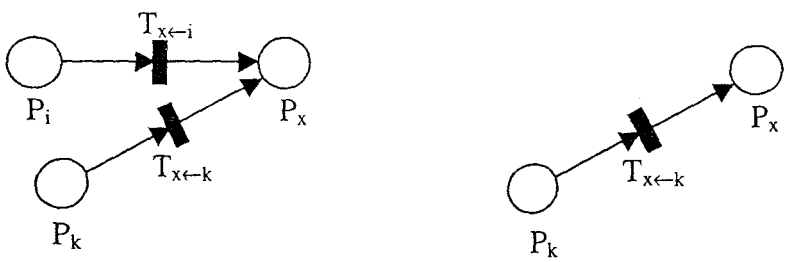
Niech $PN(P, N, F)$ będzie siecią i niech $\{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\} \cup \{p_k \rightarrow t_{x \leftarrow k} \rightarrow p_x\} \in PN$ (rys. 4.7, z lewej). **Zasłonięciem $P_i - P_x$ przez P_k** nazywamy taką transformację sieci PN w wyniku której powstaje sieć PN', w której usunięto operację $\{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$ (rys. 4.7 z prawej):

$$PN' = PN - \{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$$

PRZYKŁAD 4.7

Zasłonięcie $P_i - P_x$ przez P_k odpowiada usunięciu linii (*) z następującego fragmentu programu:

```
aux = i;        (*)
aux = k;
```



Rys. 4.7. Transformacja zasłonięcia $P_i - P_x$ przez P_k

DEFINICJA 4.12

Zmienną nieistotną nazywamy taką zmienną, która pozostaje poza sferą zainteresowa użytkownika – obserwatora, i z tego powodu pozbawiona jest reprezentacji graficznej.

W ramach wymogu redukcji przestrzennej postuluje się taką zmianę realizacji algorytmu, w której nie uczestniczą zmienne nieistotne, przy zachowaniu niezmienionej semantyki algorytmu w obszarze zainteresowania użytkownika.

LEMAT 4.4

Niech *PN(P, T, F)* będzie siecią jednokrokową synchroniczną, zawierającą podsieć $O = \{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$, oraz niech zmienna reprezentowana przez p_x będzie zmienną nieistotną. Niech *PN'(P', T', F')* będzie siecią poszerzoną o operację *OZ* zależną od *O*:

$$PN' = PN \cup OZ \text{, gdzie } OZ = \{p_x \rightarrow t_{j \leftarrow x} \rightarrow p_j\}.$$

Niech *PN''(P'', T'', F'')* będzie siecią powstałą z *PN'* w wyniku sklejenia $P_i - P_j$ wokół P_x:

$$PN'' = PN' - \{p_x \rightarrow t_{j \leftarrow x} \rightarrow p_j\} \cup \{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$$

1. Sieć *PN''* z punktu widzenia semantyki programu jest równoważna sieci *PN'*.
2. Jeżeli operacja *OZ* nie pozostaje w sekwencji z żadną operacją sieci *PN*:

$$\forall t \in T': \quad t \neq t_{k \leftarrow j} \quad \Rightarrow \quad t^* \cap t_{k \leftarrow j}{}^* = \varnothing,$$

to sieć *PN''* jest siecią jednokrokową synchroniczną.

DOWÓD

1. Po wykonaniu operacji $O = \{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$ wartości zmiennych reprezentowanych przez P_i i P_x są tożsame, zatem nie ma semantycznej różnicy między operacją $\{p_x \rightarrow t_{j \leftarrow x} \rightarrow p_j\}$ a operacją $\{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$.
2. Operacja $\{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$ nie jest zależna od operacji *O*, nie jest też zależna od żadnej innej operacji $\Theta \in PN$, gdyż wówczas również i operacja *O* byłaby zależna od Θ, a tak nie jest, gdyż *PN* jest siecią jednokrokową synchroniczną.

LEMAT 4.5

Niech *PN(P, T, F)* będzie siecią jednokrokową synchroniczną, zawierającą podsieć $O = \{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$, oraz niech zmienna reprezentowana przez P_x będzie zmienną nieistotną. Niech *PN'(P', T', F')* będzie siecią poszerzoną o operację *OS* sekwencyjną do *O*:

$$PN' = PN \cup OS \text{, gdzie } OS = \{p_k \rightarrow t_{x \leftarrow k} \rightarrow p_x\}.$$

Niech *PN''(P'', T'', F'')* będzie siecią powstałą z *PN'* w wyniku zasłonięcia $P_i - P_x$ przez P_k:

$$PN'' = PN - O.$$

1. Sieć *PN''* z punktu widzenia semantyki programu jest równoważna sieci *PN'*.
2. Jeżeli operacja *OS* nie pozostaje w zależności z żadną operacją sieci *PN*:

$$\forall t \in T': \quad \tau(t) < \tau(t_{j \leftarrow l}) \quad \Rightarrow \quad t^* \cap {}^*t_{j \leftarrow l} = \varnothing$$

to sieć *PN''* jest siecią jednokrokową synchroniczną.

DOWÓD

1. Wykonanie operacji $OS = \{p_k \rightarrow t_{x \leftarrow k} \rightarrow p_x\}$ powoduje zatarcie poprzedniej wartości zmiennej p_x, i tym samym poprzednie nadanie wartości tej zmiennej przez operację $O = \{p_i \rightarrow t_{x \leftarrow i} \rightarrow p_x\}$ pozostaje bez wpływu na dalszy przebieg algorytmu (pomijamy oczywiście semantykę samej zmiennej p_x, gdyż jest ona nieistotna).
2. Operacja $OS = \{p_k \rightarrow t_{x \leftarrow k} \rightarrow p_x\}$ nie jest sekwencyjna w stosunku do żadnej operacji $\Theta \in PN$, gdyż wówczas również i operacja *O* byłaby sekwencyjna do Θ, a tak nie jest, gdyż *PN* jest siecią jednokrokową synchroniczną.

4.7 ALGORYTMY WIZUALIZACJI ALGORYTMÓW OPARTE O FORMALIZM SIECI PETRIEGO

W podrozdziale 4.4 przedstawiona została koncepcja maszyny projekcyjnej sterowanej przepływem danych. Spełnia ona tylko jeden z trzech postulatów sformułowanych w podrozdziale 4.2, mianowicie ilustruje przepływ danych. Wprowadzony formalizm oparty o sieci Petriego pozwala rozwinąć tę metodę w celu uzyskania algorytmu uwzględniającego wszystkie trzy postulaty. Kolejno przedstawimy zatem:

- Algorytm jednokrokowy, jednokolorowy – uwzględniający wymóg temporalnej redukcji informacji.
- Algorytm jednokrokowy, trójkolorowy – rozwinięcie poprzedniego, z uwzględnieniem wymogu przestrzennej redukcji informacji.
- Algorytmy 1,5 oraz wielokrokowy, stanowiące dalsze rozwinięcia poprzedniego.

4.7.1 ALGORYTM JEDNOKROKOWY, JEDNOKOLOROWY

Algorytm jednokrokowy, jednokolorowy czyni zadość postulatowi redukcji informacji temporalnej. Opiera się on na prostym pomyśle, by przedstawić ciągi kolejno wykonywanych operacji, które należą do tego samego *kroku synchronicznego* algorytmu (por. sekcja 4.6.2)

w taki sposób, jakby były realizowane w sposób równoległy i synchroniczny. Na mocy definicji 4.7 (str. 93) realizacja taka nie zmienia algorytmu z punktu widzenia jego semantyki.

Proponowany algorytm jest implementowany w ramach procedury *Play* (por. p. 4.4), która jest kolejno wywoływana dla każdej operacji przepływu danych wykrytej w poddawanym wizualizacji oprogramowaniu. Informacje o takich operacjach są czasowo przechowywane w dynamicznej strukturze danych L. Oto podstawowe założenia algorytmu:

- Struktura L jest początkowo pusta.
- Niezmiennikiem algorytmu jest, że wszystkie operacje przechowywane w danej chwili w strukturze L należą do pojedynczego kroku synchronicznego, zatem mogą być wizualizowane synchronicznie.
- Jeśli kolejna, zgłoszona operacja przepływu danych mieści się w ramach bieżącego kroku synchronicznego algorytmu poddawanego wizualizacji, działanie procedury *Play* ogranicza się do uzupełnienia struktury L o tę operację.
- Jeśli jednak operacja nie może zostać zaliczona do kroku, wówczas wszystkie zarejestrowane w tym kroku operacje przepływu są wizualizowane (synchronicznie), i jednocześnie usuwane ze struktury L. Wreszcie, zgłoszona operacja jest zapamiętywana w strukturze L jako pierwsza w kolejnym kroku synchronicznym algorytmu.

Pseudokod funkcji *Play* implementującej opisywaną metodę wygląda następująco:

```
Zgłoś operację przepływu (F)
{
      if (F wykracza poza bieżący krok L)
      {
            Animuj(L);
            usuń wszystkie elementy z L;
      }
      dodaj F do L;
}
```

Operację *Animuj* definiuje się w sposób następujący:

```
Animuj(L)
{
      for (każda operacja przepływu F ∈ L)
            inicjalizuj wizualizację synchroniczną operacji F;
      while (trwa którakolwiek z synchronicznych wizualizacji)
            oczekuj;
}
```

Zauważmy, że cała analiza jest wykonywana przez algorytm dynamicznie, na bieżąco, w trakcie trwania projekcji, i korzysta z tych samych danych, co podstawowy algorytm wizualizacji sterowanej przepływem danych w formie wprowadzonej w punkcie 4.4.

Przyjmując, że struktura L przechowuje reprezentację operacji przepływów w danym kroku w postaci sieci Petriego $PN(P, T, F)$, i stosując notację wprowadzoną w punkcie 4.6, otrzymujemy bardziej precyzyjny zapis algorytmu:

```
Zgłoś operację przepływu (F_{j←i})
{
      OP = {p_i → t_{j←i} → p_j};
      if (∃ t_k ∈ PN: t_{j←i} nie może być pokazane synchronicznie z t_k)
      {
            Animuj(PN);
            PN = ∅;
      }
      PN = PN ∪ OP;
}
```

Na mocy lematu 4.3 (str. 94) możemy precyzyjnie określić warunek przynależności nowej tranzycji do kroku. Ostateczna postać algorytmu wygląda więc następująco:

```
Zgłoś operację przepływu (F_{j←i})
{
      OP = {p_i → t_{j←i} → p_j};
      zależność = false;
      sekwencja = false;

      { sprawdzenie zależności i sekwencji }
      for (każda tranzycja t_{y←x} ∈ PN)
      {
            if (i == y)
                  zależność = true;
            if (j == y)
                  sekwencja = true;
      }

      if (zależność || sekwencja)
      {
            Animuj(PN);
            PN = ∅;
      }
      PN = PN ∪ OP;
}
```

Ostatnia, niewielka modyfikacja algorytmu redukcji temporalnej nasunęła się jako wnio sek z obserwacji pierwszej implementacji w ramach systemu *Daphnis*. Mianowicie, w niekto rych przypadkach spełnienie postulatu redukcji temporalnej idzie za daleko: przedstawieni

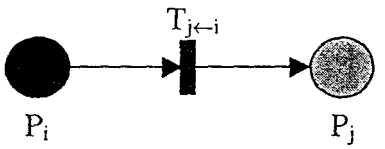
zbyt wielu operacji w tym samym czasie może czasem utrudnić zrozumienie algorytmu i zaciemnić projekcję. Niepożądany efekt występował jednak przede wszystkim przy wizualizacji iteracji. Udało się go wyeliminować wprowadzając dodatkowy warunek: jeśli kolejna, zgłoszona operacja występuje w tekście źródłowym programu wcześniej od poprzednio zgłoszonej (tj. została zrealizowana po wykonaniu instrukcji skoku wstecz), wówczas jest traktowana w taki sposób, jakby nie mieściła się w ramach bieżącego kroku synchronicznego algorytmu: powoduje animację dotychczas zarejestrowanych operacji i rozpoczyna nowy krok. Rozpoczęcie nowego kroku może też być wymuszone *a priori* przez użytkownika. Te proste poprawka rozwiązały problem.

4.7.2 ALGORYTM JEDNOKROKOWY, TRÓJKOLOROWY

Algorytm jednokrokowy, trójkolorowy stanowi pierwsze podejście do zadośćuczynienia postulatowi redukcji informacji przestrzennej. Stanowi on rozszerzenie algorytmu jednokolorowego.

Nazwa *trójkolorowy* bierze się stąd, iż korzystamy z **sieci Petriego z kolorowanymi miejscami**. Koncepcji tej nie należy mylić z kolorowanymi sieciami Petriego, badanymi między innymi przez K. Jensena [151, 157]. W jego pracach kolorowanie służyło rozróżnianiu żetonów w sieci; w naszej koncepcji kolorowanie umożliwia grupowanie miejsc (reprezentujących zmienne) w rozróżnialne klasy. Stosujemy trzy kolory:

- Wszystkie miejsca reprezentujące zmienne istotne malujemy na **czarno**.
- Dla każdej operacji przepływu danej podsiecią $\{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$, dla której miejsce reprezentujące zmienną źródłową p_i jest czarne lub szare, a miejsce reprezentujące zmienną docelową p_j nie jest czarne (reprezentowana przez nią zmienna jest nieistotna), miejsce reprezentujące zmienną p_j malujemy na **szaro** (rys. 4.8).



Rys. 4.8. Zasada kolorowania miejsc na szaro w algorytmie trójkolorowym

- Wszystkie pozostałe miejsca malujemy na **biało**.

W ramach algorytmu, po ustaleniu kolorów miejsc i po warunków sekwencji i zależności, dla wszystkich operacji $OP = \{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\}$ rozpatrujemy następujące przypadki:

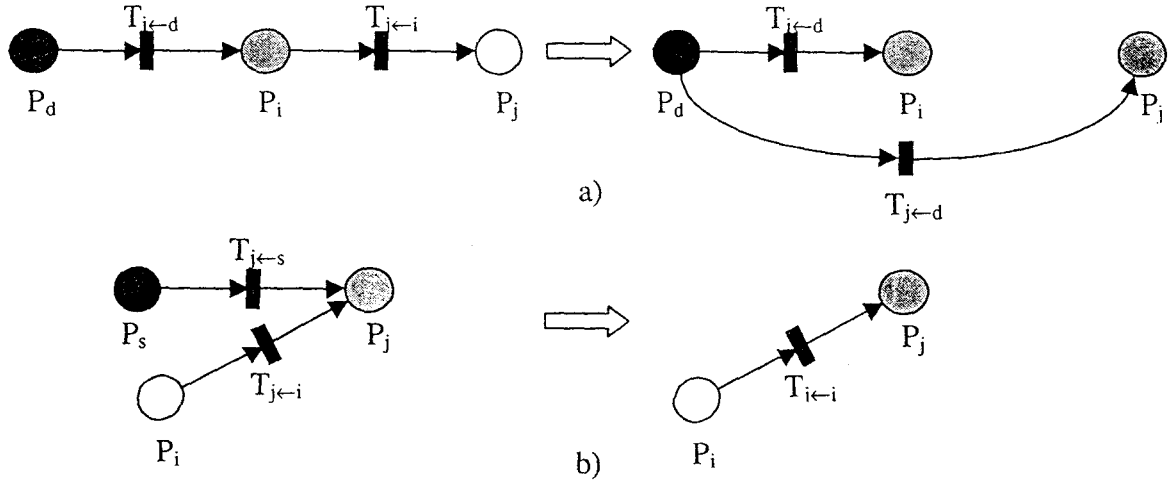
- Zmienna źródłowa reprezentowana przez p_i jest pomalowana na szaro (rys. 4.9a). Z reguły malowania na szaro wynika, że istnieje operacja $OZ = \{p_d \rightarrow t_{i \leftarrow d} \rightarrow p_i\}$, z którą operacja O jest w relacji zależności. Na mocy lematu 4.4 (str. 97), dokonujemy transformacji sklejenia $p_d - p_j$ wokół p_i (rys. 4.9a):

$$PN := PN - \{p_i \rightarrow t_{j \leftarrow i} \rightarrow p_j\} \cup \{p_d \rightarrow t_{j \leftarrow d} \rightarrow p_j\}$$

Na mocy tego samego lematu, jeśli operacja *OP* nie znajduje się w relacji sekwencji z inną operacją, uznajemy, że należy ona do bieżącego kroku algorytmu.

- Zmienna docelowa reprezentowana przez p_j jest pomalowana na szaro. Z reguły malowania na szaro wynika, że istnieje operacja $OS = \{p_s \rightarrow t_{j \leftarrow s} \rightarrow p_j\}$, z którą operacja *OP* jest w relacji sekwencji. Na mocy lematu 4.5 (str. 97), dokonujemy transformacji zasłonięcia $p_s - p_j$ przez p_i (rys. 4.9b):

$$PN := PN - \{p_s \rightarrow t_{j \leftarrow s} \rightarrow p_j\}$$



Rys. 4.9. Przypadki specjalne w algorytmie trójkolorowym i sposób ich redukcji

Na mocy tego samego lematu, jeśli operacja *OP* nie znajduje się w relacji zależności z inną operacją, uznajemy, że należy ona do bieżącego kroku algorytmu.

Po rozpatrzeniu powyższych przypadków szczególnych, przetwarzamy zgłoszoną operację *OP* w identyczny sposób, jak w algorytmie jednokolorowym.

Zapis algorytmu w pseudokodzie jest następujący:

```
Zgłoś operację przepływu (Fj←i)
{
      OP = {pi → tj←i → pj};
      zależność = false;
      sekwencja = false;
```

```
        // sprawdzenie zależności i sekwencji
        for (każda tranzycja t_{y←x} ∈ PN)
        {
              if (i == y)
              {
                    d = x; zależność = true;
              }
              if (j == y)
              {
                    s = x; sekwencja = true;
              }
        }

        // ustalenie kolorów
        if (p_i jest w obszarze zainteresowania obserwatora)
              C_i = CZARNY;
1:      else if (zależność)
              C_i = SZARY;
        else
              C_i = BIAŁY;

        if (p_j jest w obszarze zainteresowania obserwatora)
              C_j = CZARNY;
2:      else if (sekwencja)
              C_j = SZARY;
        else
              C_j = BIAŁY;
        assert(C_j != CZARNY ⇒ C_j == SZARY || !sekwencja);     (*)

        // sklejenie p_d - p_j wokół p_i na mocy lematu 4.4
        if (C_i == SZARY)
        {
              OP = {p_d → t_{j←d} → p_j};
              C_i = CZARNY;
3:            zależność = false;
        }
        assert(Ci == BIAŁY || Ci == CZARNY);                    (**)

        // zasłonięcie p_s - p_j przez p_i na mocy lematu 4.4
        if (C_j == SZARY)
        {
              PN = PN - {p_s → t_{j←s} → p_j};
              sekwencja = false;
        }
        assert(C_j != CZARNY ⇒ !sekwencja); // por. (*)         (***)

4:      if (C_i == BIAŁY && C_j != CZARNY)
              return;
        assert(C_i == CZARNY || C_j == CZARNY);// por. (**)     (****)

5:      if (zależność || sekwencja)
        {
              Animuj(PN);
              PN = ∅;
        }
        PN = PN ∪ OP;
  }
```

UWAGI:

1. c_i i c_j oznaczają poprzednie kolorowanie sieci, to znaczy nieuwzględniające operacji *OP*. Stąd możliwe jest, że: c_i == CZARNY && c_j == BIAŁY.
2. W linii (4) nie dopuszczamy do zachowania operacji, które nie będą miały wpływu na przebieg wizualizacji. Dzięki temu, jeśli operacje te były zależne lub sekwencyjne, to unikamy zbyt szybkiego i niepotrzebnego zakończenia kroku,
3. Z poprzedniego punktu wynika, że tworzony model sieci nie zawiera operacji, dla których zmienne docelowe są pomalowane na biało. Zatem:
   - Zmienna źródłowa operacji zależnej jest albo czarna, albo szara.
   - Zmienna docelowa operacji sekwencyjnej jest albo czarna, albo szara.

   Powyższe stwierdzenia pozwoliły na użycie prostszego zapisu w liniach (1) i (2).

MOŻLIWOŚĆ ULEPSZENIA:

Zauważmy, że w linii (5) spełnione są następujące asercje (por. linie (***) i (****) oraz poprzednie asercje w tekście algorytmu):

```
assert(c_i == CZARNY || c_j == CZARNY);                    (***)
assert(c_j != CZARNY ⇒ !sekwencja);                        (****)
```

Wszystkie operacje przepływu danych można ze względu na wartości c_i i c_j podzielić na dwie klasy:

1. c_j == CZARNY – są to operacje, które mają bezpośredni wpływ na przebieg projekcji, i nie będziemy dla nich wprowadzać żadnych zmian.
2. c_i == CZARNY && c_j != CZARNY – operacje tej klasy mogą, ale nie muszą mieć wpływu na przebieg wizualizacji (jest to uzależnione od tego, czy wystąpi operacja zależna, co umożliwiłoby przeprowadzenie transformacji sklejenia).

Możliwość ulepszenia algorytmu wynika ze spostrzeżenia, że operacje klasy (2), pozostające w relacji zależności (na mocy asercji (****) na pewno nie pozostają w relacji sekwencji), o ile nie zostaną wykorzystane w ramach procedury sklejania, zupełnie nadmiarowo powodują zakończenie bieżącego kroku synchronicznego algorytmu. W związku z tym, można wprowadzić je do sieci *PN* bezwarunkowo, bez wymuszania zakończenia bieżącego kroku synchronicznego. W konsekwencji, w linii (3) musimy dodatkowo sprawdzić, czy operacja $\{P_d \rightarrow T_{i \leftarrow d} \rightarrow P_i\}$ nie jest przypadkiem zależna.

Rezultatem ulepszenia algorytmu jest modyfikacja następujących dwóch linii (etykiety odpowiadają tekstowi pseudokodu powyżej):

```
3:      zależność = zależność({Pd → Ti←d → Pi});
5:      if (Cj == CZARNY && (zależność || sekwencja))
```

W realnej implementacji informacja o relacji zależności operacji, z której korzysta się w linii (3), może być przechowywana w postaci flagi binarnej w ramach reprezentacji *PN*.

4.7.3 ALGORYTM: 1,5-KROKOWY I WIELOKROKOWY

Transformacja sklejania pozwala na wieloetapowe śledzenie wartości danych, przepływających przez zmienne nieistotne, i uwidocznienie pośrednich zależności między zmiennymi istotnymi. Jednak każdorazowo po zakończeniu analizy bieżącego kroku synchronicznego algorytmu sieć reprezentująca graf przepływów jest zerowana: $PN = \varnothing$. Ogranicza to możliwości wykrycia takich pośrednich zależności do pojedynczego kroku synchronicznego. Stąd nazwa algorytmu: *jednokrokowy*. Istnieją dwa podejścia przełamujące to ograniczenie:

- **Algorytm 1,5-krokowy**. Część operacji przepływu danych jest zachowywana w sieci *PN* po zakończeniu kroku. Dotyczy to tylko tych operacji przepływu, w których zmienna źródłowa nie była modyfikowana, to jest takich, dla których reprezentująca je tranzycja t_z spełnia warunek: $\forall t \in PN: \; t^* \cap {}^*t_z = \varnothing$. Łatwo zauważyć, że wartości zmiennych źródłowej i docelowej są wtedy tożsame, również po zakończeniu kroku, a zatem nie ma przeszkód, by zastosować ewentualną transformację sklejenia. Algorytm przełamuje więc horyzont pojedynczego kroku, wprawdzie z pewnymi ograniczeniami, ale za to konieczne modyfikacje w stosunku do wersji jednokrokowej są niewielkie.
- **Algorytm wielokrokowy** jest dalszym rozwinięciem tego wątku: nawet jeśli zmienna źródłowa została zmodyfikowana, w dalszym ciągu możemy przeprowadzić wizualizację sklejonego przepływu, jeśli rozciągniemy ją w czasie. Ciągły ruch obiektu graficznego powinien rozpocząć się zanim zmieniona zostanie wartość zmiennej docelowej, a zakończyć w chwili faktycznego wartościowania zmiennej docelowej, przy czym obejmować może wiele kolejnych kroków. Jeśli realizacja nie narzuci ograniczeń na liczbę kroków, w trakcie której śledzić można złożoną, wieloetapową operację przepływu danych, wówczas algorytm ten pozwoli na zobrazowanie wszystkich pośrednich zależności między zmiennymi. Takie podejście może jednak łatwo okazać się zbyt trudne pojęciowo dla obserwatora; poza tym stopień komplikacji algorytmu staje się bardzo znaczny.

W systemie *Daphnis* nie pokusiliśmy się o implementację algorytmu wielokrokowego. Poprzestaliśmy na algorytmie 1,5-krokowym, wierząc, że nie jest on znacząco gorszy,

a wręcz przeciwnie, jest kompromisem pomiędzy postulatem jak najszerszego przedstawienia zależności między danymi, a z drugiej strony – utrzymaniem prostoty i przejrzystości prezentacji. Poniżej zamieszczono algorytm 1,5-krokowy, trójkolorowy, w postaci zapisu w pseudokodzie:

```
Zgłoś operację przepływu (F_{j←i})
{
        ...
        // wszystkie fragmenty pozostają bez zmian
        // z wyjątkiem końcowej części.

5:      if (C_j == CZARNY && (zależność || sekwencja))
        {
              Animuj(PN);
              for (każda operacja OX = {p_x → t_{y←x} → p_y} ∈ PN)
                    if (p_y jest w obszarze zainteresowania obserwatora
                    || ∃ t∈T: t*∩*t_{y←x} ≠ ∅)
                          PN = PN - OX;
        }
        PN = PN ∪ OP;
  }
```

4.8 UWAGI KOŃCOWE

Analizując różne sposoby opisu oprogramowania sformułowaliśmy trzy postulaty skutecznej wizualizacji algorytmów. Są to:

- Postulat wizualizacji przepływu danych,
- Postulat redukcji informacji przestrzennej,
- Postulat redukcji informacji temporalnej.

Uważamy, że spełnienie powyższych wymagań pozwala wyekstrahować z poddawanego wizualizacji oprogramowania informacje niezwykle ważne dla zrozumienia sposobu działania algorytmów, równocześnie maskując informacje mniej istotne. Wykazaliśmy, że ich spełnienie podwyższa poziom abstrakcji wizualizacji. Większość animacji algorytmów stworzonych przy pomocy istniejących dotychczas systemów w ten czy inny sposób spełniała wymienione postulaty. Wymagało to jednak ręcznego projektowania i strojenia projekcji, a nierzadko też daleko idących modyfikacji tekstu źródłowego poddawanych wizualizacji programów. Pojawia się zatem potrzeba stworzenia systemu, który w sposób automatyczny uwzględniałby wszystkie trzy postulaty w tworzonych wizualizacjach. System taki miałby zapewnioną znaczną przewagę nad istniejącymi dotychczas rozwiązaniami.

Zaproponowaliśmy oryginalny algorytm animacji algorytmów, korzystający z techniki śledzenia przepływu danych i oparty na pewnych przekształceniach sieci Petriego te przepływy odzwierciedlających. Jedną z jego zaawansowanych wersji, zwaną **algorytmem 1,5-krokowym, trójkolorowym**, zastosowaliśmy przy implementacji systemu animacji algorytmów o nazwie *Daphnis*. Tworzone z pomocą tego systemu w sposób półautomatyczny projekcje spełniają wszystkie trzy postulaty, nawet, jeśli użytkownik włożył w ich przygotowanie relatywnie mało wysiłku – w porównaniu z dotychczas istniejącymi systemami. Wizualizator poszukujący tak zwanego „szybkiego i łatwego narzędzia" może być usatysfakcjonowany już pierwszymi wynikami pracy z systemem. Użytkownicy bardziej wymagający otrzymują od razu projekcję charakteryzującą się stosunkowo wysokim poziomem abstrakcji, która jest punktem wyjściowym do stworzenia animacji bardziej zaawansowanych.

Zalety przedstawionego algorytmu zostały w pełni potwierdzone doświadczeniami uzyskanymi podczas pracy z systemem *Daphnis*. Wizualizacje dość dobrze oddające semantykę algorytmu tworzy się szybko i łatwo. Stosunkowo nieskomplikowane jest również projektowanie animacji bardziej zaawansowanych. Wiele przykładów takich wizualizacji przedstawimy w rozdziale 8.

5
OGÓLNA CHARAKTERYSTYKA SYSTEMU DAPHNIS

Opracowanie metody animacji algorytmów opartej na śledzeniu przepływu danych jest w naszym przeświadczeniu najważniejszym dokonaniem teoretycznym w ramach niniejszej pracy. Jednak jej praktycznym efektem jest stworzenie systemu animacji algorytmów o nazwie *Daphnis*. W niniejszym rozdziale przedstawiono ogólną charakterystykę systemu; kolejne rozdziały poświęcamy, odpowiednio, środowisku interakcyjnemu systemu i szczegółom związanym z jego implementacją.

5.1 GENEZA SYSTEMU *DAPHNIS*

Nazwa naszego systemu jest swobodnie potraktowanym skrótem wyrażenia *DAta-Flow-driven aNImation System.*

System *Daphnis* jest następcą poprzednich systemów animacji algorytmów, jakie powstały w toku badań nad wizualizacją prowadzonych w Zakładzie Oprogramowania Instytutu Informatyki Politechniki Śląskiej, w których brał udział autor niniejszej pracy. Pierwszy system, *SANAL* [103 – 105], był rozwijany w latach 1989-1996 i stał się poligonem doświadczalnym, który pozwolił zebrać i sformułować doświadczenia niezbędne dla stworzenia kolejnych produktów. Inspiracją dla pierwszych twórców systemu *SANAL* był system *BALSA* Marca H. Browna [73 – 75]. System *SANAL* pracował na komputerach klasy *IBM PC*, pod kontrolą systemu operacyjnego *MS-DOS*.

W 1993 roku powstała wersja *3.0* systemu *SANAL*, poszerzona o możliwość realizacji synchronicznej projekcji procesu wykonania kilku algorytmów równocześnie [105, 106]. W toku prac okazało się jednak, że istniejąca architektura systemu nie jest już wystarczająca. W szczególności konieczne było odejście od zastosowanej w nim metody sterowania projek-

cją za pomocą zdarzeń, oraz zastosowanie architektury obiektowej (przy czym swego rodzaju obiektowa „proteza" została zrealizowana w ramach wersji *3.0*). Tak więc następny system, *WinSANAL* [107 – 110, 114], opracowany w latach 1995-1997, nie był tylko prostym przeniesieniem *SANAL*-a do środowiska *Windows*, jak mogłaby sugerować jego nazwa. Stanowił zupełnie nowe rozwiązanie, stworzone od podstaw. *WinSANAL* był systemem sterowanym danymi, o architekturze w pełni obiektowej. System ten nigdy nie wyszedł poza wersję prototypową.

Osobiste doświadczenia autora niniejszej pracy w dziedzinie implementacji systemów wizualizacji oprogramowania sięgają roku 1993, kiedy zrealizował rozszerzenia systemu *SANAL*, których efektem było stworzenie wersji 3.0. Był też pomysłodawcą, głównym projektantem i wykonawcą systemu *WinSANAL*. Obecny system, *Daphnis*, stanowi jego oryginalne osiągnięcie.

Prace nad systemem *Daphnis* trwają od 1997 roku. Jest on rozwiązaniem zupełnie nowym, choć wiele jego elementów nawiązuje do systemu *WinSANAL*, a nawet wykorzystano w nim niektóre fragmenty tekstu źródłowego. System *Daphnis* odróżnia od poprzednika przebudowana architektura wewnętrzna. Najważniejszą innowacją jest zastosowanie metody animacji algorytmów bazującej na śledzeniu przepływu danych. Umożliwia ona wprowadzenie istotnych udogodnień dla użytkownika, który mniejszym nakładem pracy jest w stanie osiągnąć efektowniejsze projekcje, to jest takie, które charakteryzują się stosunkowo wysokim poziomem abstrakcji.

Ważną inspiracją przy tworzeniu *Daphnis* był też prototypowy system *Siamoa* [115 – 121], ukończony w 1997 przez F. Van de Veire'a z Uniwersytetu Lille 1 we Francji, pod kierunkiem prof. J.-M. Toulotte'a. Z systemu *Siamoa* przejęliśmy między innymi filozofię architektury *POST* [31, 32, 120].

Należy zaznaczyć, że system *Daphnis* powstawał jako system prototypowy i w dalszym ciągu – w pewnym stopniu – zachował taki charakter. Jego głównym przeznaczeniem jest wypróbowanie w praktycznej implementacji nowej metody wizualizacji algorytmów. Jesteśmy przekonani, że w zakresie realizacji projekcji algorytmu i interaktywnego środowiska pracy użytkownika obserwującego tę projekcję dysponujemy już sprawnym i w pełni funkcjonującym, kompletnym narzędziem. Jednak część zaplanowanych środków wspomagających pracę wizualizatora wykracza poza ramy naszej pracy, i niektóre prace implementacyjne są jeszcze w toku. Część z nich zostanie lub jest wykonywana przez naszych współpracowników, i będzie tylko zasygnalizowana w niniejszej pracy; inne fragmenty zostaną zrealizowane

w ramach projektów studenckich; są też wreszcie elementy (jak bezpośrednia manipulacja obiektami graficznymi), które były już zrealizowane w jednej z poprzednich wersji oprogramowania (*WinSANAL*), a które na skutek późniejszych prac implementacyjnych „wypadły" z systemu, i nie zostały jeszcze, jak dotąd, ponownie zaimplementowane (czy raczej zainstalowane, jako że ich implementacja istnieje, wymaga jedynie odpowiedniego scalenia z systemem w nowej architekturze).

5.2 CHARAKTERYSTYKA SYSTEMU

Daphnis jest systemem animacji algorytmów ogólnego przeznaczenia. Oznacza to, że wspomaga tworzenie wizualizacji obrazujących działanie dowolnej klasy algorytmów. System pracuje na komputerach klasy *IBM PC*, pod kontrolą systemu operacyjnego *Windows 95*, *Windows NT 4.0* lub nowszych. Minimalne wymagania sprzętowe są takie, jak wymagania odpowiednich systemów operacyjnych, zalecane jest jednak zastosowanie komputera z procesorem klasy *Pentium* lub lepszym.

System *Daphnis* jest w zasadzie niezależny od języka, w którym sformułowano algorytm poddawany wizualizacji. Poddawany wizualizacji program musi być jedynie przystosowany do pracy w 32-bitowym środowisku systemu *Windows* i mieć dostęp do procedur zaimplementowanych w postaci dynamicznie wiązanej biblioteki (ang. *dynamic-link library*, *dll*). Stworzony został interfejs programowy maszyny projekcyjnej systemu dla programów napisanych w językach *C* i *C++*, dzięki któremu przygotowanie do wizualizacji algorytmów stworzonych w tych właśnie językach jest szczególnie łatwe. Użycie innych języków wymaga zastosowania stosownych deklaracji. Jak dotąd, poza językami *C* i *C++*, zrealizowaliśmy wizualizacje programów napisanych w języku *Visual Basic* [206, 207], i nie stanowiło to poważniejszego problemu technicznego.

System *Daphnis* jest systemem sterowanym danymi, z deklaratywną metodą specyfikacji wizualizacji (por. z klasyfikacją metod sterowania w systemach wizualizacji w rozdziale 2.5). Niezbędne są jednak pewne modyfikacje tekstu źródłowego programu, tak więc styl specyfikacji jest inwazyjny. Do przeprowadzenia projekcji niezbędne jest dostarczenie dwóch elementów. Oto one:

- Odpowiednio zmodyfikowana wersja programu przeznaczonego do wizualizacji. Otrzymuje się ją poprzez skompilowanie i scalenie (zlinkowanie) tekstu źródłowego programu, wzbogaconego o niezbędne odwołania systemowe.

- Skrypt konfiguracyjny – plik tekstowy, stanowiący zapis deklaratywnej specyfikacji wizualizacji, szczegółowo definiującej jej zakres i formę.

Poddany modyfikacjom i skompilowany program może być wywoływany w połączeniu z różnymi skryptami konfiguracyjnymi, bez potrzeby powtórnej kompilacji. Uzyskiwać można w ten sposób projekcje istotnie różniące się swoim zakresem, formą, a także poziomem abstrakcji. Ogranicza to wyraźnie inwazyjność systemu – przynajmniej z punktu widzenia użytkownika.

Uruchomienie przygotowanego do wizualizacji programu powoduje otwarcie **okna projekcyjnego**, w którym przeprowadzana jest wizualizacja algorytmu. Okno to stanowi interfejs użytkownika – obserwatora projekcji, i pozwala mu na interakcję z systemem – zatrzymanie i wznowienie projekcji, regulację tempa projekcji i skali obrazu, a także dokonywanie inspekcji danych. Okno projekcyjne jest otwierane i obsługiwane niezależnie od jakichkolwiek innych okien, które otwierałby poddawany wizualizacji program. Możliwa jest też wizualizacja programów pozbawionych własnego interfejsu użytkownika – okno projekcyjne staje się wówczas jedynym dostrzegalnym przejawem wykonywania programu. Wszystkie przykładowe wizualizacje, dostarczane wraz z systemem, są skonstruowane w taki właśnie sposób. Program sterujący projekcją jest synchronizowany względem tempa tej projekcji[6] – podobnie, jak to ma miejsce w przypadku uruchamiania programu pod kontrolą konwencjonalnego debuggera.

5.3 ELEMENTY SYSTEMU *DAPHNIS*

Najistotniejszą częścią systemu *Daphnis* są moduły niezbędne do aktywowania i przeprowadzenia projekcji, co się sprowadza do uruchomienia programu przygotowanego do wizualizacji. Łącznie tworzą one **maszynę projekcyjną** systemu.

- Moduł maszyny projekcyjnej, składający się z dwóch bibliotek typu *dll* (ang. *dynamic link library*): *daphnis.dll* oraz *daphnisWin.dll.* Jest on przeznaczony do bezpośredniego, dynamicznego scalenia z programami poddawanymi wizualizacji. Zawiera wszystkie funkcje przeznaczone do bezpośredniego wywoływania z poddawanego wizualizacji programu. Można go stosunkowo łatwo scalić z programami napisanymi w niemal dowolnym języku zaimplementowanym w środowisku *Windows*. Mechanizmy zawarte w module

[6] Ta cecha systemu *Daphnis* ogranicza jego przydatność do wizualizacji procesów czasu rzeczywistego. W tym celu należałoby system rozbudować o możliwość prezentacji wizualizacji *post-mortem.*

odpowiadają za otwarcie i obsługę okna projekcyjnego, wczytanie i interpretację skryptu konfiguracyjnego oraz zestawienie na tej podstawie elementów sterujących projekcją. Zadaniem modułów bibliotecznych jest następnie przyjmowanie i szeregowanie zgłoszeń pochodzących z wizualizowanego programu, akwizycja danych przejmowanych z tego programu i wreszcie zarządzanie wyświetlaniem obrazu w obrębie okna projekcyjnego.

- Składniki zrealizowane w technologii *ActiveX* (ang. *ActiveX components*) [203, 205 – 208]. Są to biblioteki udostępniające zuniformizowane interfejsy obiektów sterujących przebiegiem projekcji. W szczególności są to translatory, przetwarzające pozyskane z poddawanego wizualizacji programu dane, oraz tzw. grele, odpowiedzialne za ostateczną reprezentację graficzną obiektów. Obiekty te współpracują z opisanym w poprzednim akapicie modułem maszyny projekcyjnej. Użytkownik może tworzyć własne składniki *ActiveX*, poszerzając w ten sposób zakres funkcjonalny maszyny projekcyjnej. Obiekty dodane przez użytkownika mogą być wykorzystywane w projekcjach na równi z obiektami predefiniowanymi.

Powyższe elementy są wystarczające zarówno do przeprowadzenia projekcji, jak i do jej przygotowania, i w tym sensie stanowią same w sobie kompletny zestaw narzędzi. Mimo to *Daphnis* poszerzono o dodatkowe moduły, ułatwiające pracę różnym grupom użytkowników. Są to:

- Zestaw przykładowych, gotowych do projekcji programów, realizujących wybrane algorytmy. Dostępne obecnie przykładowe animacje przygotowano jedynie z myślą o zademonstrowaniu możliwości systemu, lecz stanowić mogą one zalążek reprezentatywnego zestawu wizualizacji algorytmów, mogącego być istotną pomocą dydaktyczną.
- Program o nazwie *DAPHNIS.EXE*, ułatwiający aktywizację wybranej projekcji. Umożliwia on uruchomienie jednej z wymienionych wyżej animacji przykładowych, jak również własnej projekcji użytkownika.
- Środki wspomagające pracę wizualizatora: preprocesor tekstu źródłowego programu oraz narzędzia interaktywne ułatwiające przygotowanie skryptów konfiguracyjnych. W chwili obecnej środki te są na etapie realizacji.

6
ŚRODOWISKO INTERAKTYWNE SYSTEMU DAPHNIS

Środowisko interaktywne systemu *Daphnis* zostało zaprojektowane w taki sposób, by spełnić wymogi użytkowe stawiane przez różnorodne grupy użytkowników. Jego opis poprzedzimy zatem przeglądem potrzeb i wymagań potencjalnych użytkowników systemu. Sformułujemy podstawowe postulaty projektowe dotyczące sfery interfejsu użytkownika. Następnie przedstawimy środowisko pracy zarówno użytkownika będącego obserwatorem projekcji, jak i wizualizatora.

6.1 ANALIZA POTRZEB I WYMAGAŃ UŻYTKOWNIKÓW

Nie sposób poprawnie zaprojektować środowiska interaktywnego, a w szczególności interfejsu użytkownika, nie dokonując uprzednio analizy wymagań i potrzeb potencjalnych użytkowników systemu. Dopiero wtedy można właściwie sformułować wymogi użytkowe systemu i zaprojektować sferę pośrednictwa użytkownika w sposób racjonalny, przemyślany i optymalny.

W przypadku systemu animacji algorytmów, jakim jest Daphnis, w grę wchodzą różne grupy użytkowników, z których każda reprezentuje inny zakres potrzeb i wymagań. Ze względu na rodzaj interakcji, w jakie wchodzą oni z systemem, możemy wyróżnić (zgodnie z klasyfikacją przedstawioną w początkowej części rozdziału 2):

- **programistów**, którzy tworzą programy przeznaczone do wizualizacji,
- **wizualizatorów** (**animatorów**), którzy przygotowują je do projekcji,
- **obserwatorów**, którzy nie muszą ograniczać się do oglądania prezentacji algorytmu w działaniu, ale mogą też wpływać na przebieg projekcji.

Użytkownicy z grupy programistów korzystają z oddzielnego środowiska przeznaczone do projektowania programów, tym samym nie są w ścisłym sensie użytkownikami nasze systemu, i nie będą przedmiotem dalszej analizy.

Ze względu na rodzaj zastosowań systemu, zgodnie z podziałem wprowadzonym w r dziale 2.7, wyróżniamy użytkowników związanych z dydaktyką informatyki oraz z inżynie oprogramowania. W obrębie pierwszej grupy naturalny jest dalszy podział na **uczniów** i **uczycieli**. Użytkowników drugiej grupy zastosowań nazwiemy krótko **projektantami**.

Podsumowując, klasyfikacji użytkowników systemu możemy dokonać w dwóch wym rach: pod względem rodzaju interakcji z systemem (obserwatorzy i wizualizatorzy) oraz p względem sposobu zastosowania systemu (uczniowie, nauczyciele i projektanci). Daje w efekcie siatkę obejmującą sześć różnych kategorii (tab. 6.1). Uczniowie w sposób natura mogą być zarówno obserwatorami (widzami), jak i wizualizatorami. Ci ostatni to uczniow eksperymentatorzy, którzy samodzielnie dokonują eksploracji algorytmów. Głównym w kiem działalności nauczycieli jest przygotowywanie demonstracji – wizualizacji algorytm przeznaczonych dla potrzeb dydaktycznych. Obserwatorami takich prezentacji mają uczniowie, przeto grupy nauczycieli-obserwatorów nie będziemy analizować. Co do pro tantów, korzystają oni z systemu wizualizacji jak ze swego rodzaju „wizualnego *debugge* W ich przypadku role wizualizatora i obserwatora są ściśle sprzężone, przeto rozpatrywa będziemy razem. Oczywiście, indywidualni użytkownicy mogą wchodzić w różne role. przykład wykładowca uniwersytecki może, jako nauczyciel, stosować system do celów daktycznych, równocześnie używając go jako narzędzia inżynieryjnego w swoich pracach dawczo-projektowych i w tym samym czasie poszerzać swoją wiedzę analizując gotowe zualizacje zaawansowanych algorytmów, w charakterze użytkownika – ucznia.

	OBSERWATOR	WIZUALIZATOR
UCZEŃ	uczeń – widz	uczeń – eksperymentat
NAUCZYCIEL		nauczyciel przygotowuj demonstrację dydaktycz
PROJEKTANT	projektant stosujący system jako „wizualny *debugger*"	

Tab. 6.1. Zestawienie kategorii użytkowników rozpatrywanych w systemie *Daphnis*

6.1.1 UCZEŃ – WIDZ

Kategoria uczniów obejmuje użytkowników o bardzo różnych poziomach kompetencji. Mogą to być uczniowie i studenci rozpoczynający naukę programowania, studenci późniejszych lat studiów informatycznych, analizujący działanie bardziej złożonych algorytmów, a nawet projektanci, korzystający z systemu celem poszerzenia swej wiedzy w zakresie algorytmiki. System może być także wykorzystywany do prezentacji nowo opracowanych algorytmów, na przykład w ramach konferencji i seminariów naukowych; faktycznie więc zakres kompetencji grupy użytkowników nazwanej *uczniami* rozciąga się od zupełnych nowicjuszy aż po doświadczonych projektantów, programistów i wykładowców uniwersyteckich – co jest zgodne z zasadą, że człowiek uczy się przez całe życie.

W najprostszym przypadku użytkownik należący do grupy uczniów-widzów oczekuje od systemu tylko tego, że przedstawi mu on projekcję algorytmu w sposób nie wymagający z jego strony znaczniejszego zaangażowania. Nie należy oczekiwać, że użytkownik będzie obeznany z systemem lub że zechce on włożyć zbyt wiele wysiłku w zrozumienie trybu działania lub zapoznanie się z instrukcją obsługi systemu. Uruchomienie projekcji wybranego algorytmu (programu) powinno być bardzo łatwe, mniej więcej tak, jak otwarcie dokumentu w przeciętnej aplikacji biurowej. Również narzędzia, za pomocą których użytkownik może wpływać na przebieg projekcji, muszą być zaprojektowane w sposób przejrzysty i oczywisty. Wyróżnić tu należy następujące elementy:

- Sterowanie przebiegiem projekcji: jej rozpoczęcie, przerwanie, wznowienie i zakończenie, praca w trybie krokowym, a także regulacja tempa pokazu.
- Sterowanie obrazem: przede wszystkim skalowanie i przewijanie obrazu.
- Interakcja z poddawanym wizualizacji programem: zmiana zawartości lub układu obserwowanych struktur danych (elementy wizualnego *debuggera*).

Zauważmy, że ostatnia z wymienionych potrzeb dotyczy bardziej zaawansowanych użytkowników, bliższych raczej grupie uczniów – eksperymentatorów.

Z punktu widzenia ucznia problemem jeszcze ważniejszym, niż interfejs użytkownika, jest zawartość graficzna i semantyczna projekcji: zasób środków graficznych, sposób odwzorowania algorytmu i inne elementy ułatwiające zrozumienie lub wyjaśniające działanie algorytmu. Za krytyczne uznajemy trzy czynniki:

- **Zrozumiałość i klarowność**. Niezrozumiała, niepotrzebnie zawiła projekcja mija się z celem. Graficzna reprezentacja przedstawianego algorytmu musi być trafna i łatwa do

skojarzenia. Istotna jest adekwatność stosowanych metafor. Powinny one odwoływać się do wzorców i pojęć znanych obserwatorowi, a zatem albo nawiązujących do świata rzeczywistego, albo też do powszechnie stosowanych schematów, takich, jakie spotkać możemy na przykład w podręcznikach algorytmiki.

- **Zawartość informacyjna i wysoki poziom abstrakcji**. Samo odwzorowanie, nawet dynamiczne, występujących w programie danych nie jest wystarczające. Potrzebne są elementy wskazujące na różnorodne niuanse związane z działaniem algorytmu. Dla przykładu, wizualizacja algorytmu sortowania nie może się ograniczać do prezentacji kolejnych zmian pozycji sortowanych elementów, ale musi też uwidaczniać najważniejsze dla danego algorytmu operacje, takie, jak porównywanie wartości, wyszukiwanie elementu ekstremalnego czy wyznaczanie mediany. Przydatne są też widoki syntetyczne, przedstawiające dane w postaci wysoce przetworzonej: na przykład historie wykonania algorytmu czy zestawienia statystyczne. Ważne są również dodatkowe elementy ułatwiające zrozumienie, stanowiące rodzaj komentarza, jak choćby podpisy wyjaśniające znaczenie poszczególnych elementów obrazu.
- **Atrakcyjność formy**. Wartość nawet bardzo poprawnej i bogatej koncepcyjnie projekcji będzie ograniczona, jeśli nie zostanie ona podana w takiej formie, by przykuć uwagę użytkownika i pobudzić jego zainteresowanie. Wizualizacja powinna być estetyczna, żywa, odznaczać się różnorodnością użytych form graficznych, celowym i wyważonym zastosowaniem koloru, płynnej animacji, a ewentualnie także i dźwięku.

Podsumowując, ze względu na ucznia – widza, interfejs użytkownika powinien być prosty i funkcjonalny, tym bardziej, że wymagania użytkownika nie są zbyt rozległe. Natomiast należy zapewnić bardzo bogaty i obszerny repertuar środków graficznych.

6.1.2 UCZEŃ - EKSPERYMENTATOR

Od użytkownika określanego jako uczeń – eksperymentator możemy wymagać wyższego poziomu kompetencji, niż w przypadku poprzednio omawianej grupy. Wyróżniamy trzy główne rodzaje aktywności uczniów – eksperymentatorów:

- Interakcja z obserwowaną wizualizacją: zmiana zawartości lub układu obserwowanych struktur danych (tę aktywność wymieniliśmy też omawiając uczniów – widzów).
- Wprowadzanie poważniejszych modyfikacji do wizualizacji przygotowanych wcześniej przez nauczyciela.

- Tworzenie nowych wizualizacji własnych algorytmów.

Interakcja z obserwowaną wizualizacją, czyli zastosowanie systemu jako „wizualnego *debuggera*", jest najprostszym dla użytkownika sposobem eksperymentowania z algorytmami. Ponieważ dla wielu użytkowników będzie to pierwsza czynność, na jaką się odważą po biernej obserwacji projekcji, a dla części z nich będzie to zapewne już najbardziej zaawansowane doświadczenie, na jakie sobie pozwolą, musimy je uczynić tak prostym, jak to tylko możliwe. Eksperymentator powinien w łatwy sposób otrzymać tekstową reprezentację wartości danych wyświetlanych w postaci graficznej, i za pomocą prostego dialogu być w stanie zastąpić ją inną wartością. Bardziej zaawansowana technicznie, za to łatwiejsza do opanowania przez użytkownika byłaby metoda zmiany wartości oparta na intuicyjnie zrozumiałym mechanizmie bezpośredniej manipulacji obiektami graficznymi na ekranie, za pomocą myszy.

Od użytkowników modyfikujących istniejące wizualizacje możemy, a nawet musimy wymagać pewnej wiedzy na temat sposobu przygotowania wizualizacji w systemie. W dalszym ciągu istotna jest prostota i łatwość użycia elementów interfejsu użytkownika; również przejrzysty i prosty sposób specyfikowania wizualizacji może stanowić zachętę do dalszych eksperymentów. Jeśli składnia i styl języka specyfikacji będą dostatecznie jasne, a być może nawet intuicyjnie wyczuwalne, użytkownik może dokonywać drobnych modyfikacji w istniejących specyfikacjach nawet bez uprzedniego przygotowania w zakresie znajomości obsługi systemu, na bieżąco ucząc się poprzez analizę istniejącego zapisu.

W przypadku uczniów tworzących swoje własne wizualizacje, ich kompetencje i wymagania są zbliżone do kompetencji i wymagań nauczycieli, i nie będziemy ich w tym miejscu osobno analizować. Ponadto uczniowie – eksperymentatorzy mogą korzystać z ułatwień przeznaczonych dla użytkowników – projektantów. Powinny one pozwalać na tworzenie niezbyt wymyślnych wizualizacji w bardzo krótkim czasie, a nawet w pełni automatyzować ten proces.

Uczniowie – eksperymentatorzy stanowią grupę, której wymagania i oczekiwania względem systemu plasują się pomiędzy wymaganiami uczniów – widzów a wymaganiami nauczycieli.

6.1.3 NAUCZYCIEL

Celem użytkownika – nauczyciela jest tworzenie projekcji algorytmów dla celów dydaktycznych. Łatwość i prostota obsługi systemu nie jest tu czynnikiem aż tak krytycznym, jak to miało miejsce w przypadku korzystania z niego przez uczniów. Obowiązuje zasada: niezbyt skomplikowane wizualizacje powinny być relatywnie proste w realizacji, zaś tworzenie wy-

myślnych animacji na wysokim poziomie abstrakcji, z dużą zawartością informacji dodatkowych (tzw. zawartości intencyjnej – por. rozdział 1.2.2) może być trudniejsze i wymagać wcześniejszego przygotowania w zakresie znajomości obsługi systemu. W istocie, ze wszystkich omawianych grup użytkowników, nauczyciele wymagają najwięcej, gdyż to właśnie oni korzystają z pełni możliwości systemu w zakresie tworzenia zaawansowanych koncepcyjnie i technicznie wizualizacji – i *vice versa*, od nauczycieli możemy spodziewać się najwyższego poziomu kompetencji w zakresie umiejętności obsługi systemu.

Zgodnie z nieco żartobliwą klasyfikacją wprowadzoną przez P. Coada i E. Yourdona [188], oddziaływanie systemu oprogramowania na użytkownika może obejmować następujące zasadnicze typy reakcji:

- Przerażenie, gniew, irytacja, zakłopotanie,
- Znudzenie,
- Przypływ sił twórczych, ożywienie.

W przypadku systemów wizualizacji algorytmów, użytkownicy – obserwatorzy, oglądający przebieg projekcji, reagują zwykle zgodnie z ostatnim schematem. Wynika to z atrakcyjności sposobu przedstawiania treści. Niestety, użytkownicy próbujący samodzielnie tworzyć wizualizacje bardzo często doznają emocji bliskich pierwszej, a w najlepszym przypadku drugiej kategorii. Jednym z ciekawych, i jakże inspirujących kontrprzykładów były organizowane przez M.H. Browna w latach 1992-93 festiwale animacji algorytmów [45, 46], w których osoby zaproszone do samodzielnego wizualizowania algorytmów doświadczały entuzjastycznego przypływu sił twórczych. Niestety, to tylko stosunkowo rzadki wyjątek od reguły.

Jak wielokrotnie deklarowaliśmy, celem tej pracy jest ułatwienie i, co za tym idzie uprzyjemnienie pracy wizualizatora, który może otrzymać ciekawe prezentacje bez żmudnego, ręcznego ich programowania, niezbędnego w wielu innych rozwiązaniach. Aby jednak efekt naszych wysiłków był dostrzegalny w warunkach rzeczywistych, musimy zadbać o całokształt aspektów wpływających na komfort pracy użytkownika.

Poniżej wymienimy czynniki, które uznajemy za krytyczne dla nauczycieli:

- Stosunkowo jasny, przejrzysty język i styl zapisu specyfikacji tworzonych wizualizacji.
- Wprowadzenie narzędzi ułatwiających pracę, takich jak preprocesory tekstu źródłowego czy edytory graficzne pozwalające w naturalny, interaktywny sposób projektować układ graficzny przyszłej prezentacji.

- Łatwość przejścia od fazy specyfikowania wizualizacji do fazy obserwacji projekcji. Kluczowe w tej mierze jest, aby większość prac nad tworzeniem wizualizacji następowała poza tekstem źródłowym programu, i wprowadzenie zmian nie wymagało każdorazowego kompilowania i scalania programu poddawanego wizualizacji.
- Integracja różnych narzędzi dostępnych wizualizatorowi i możliwie jak najlepszy interfejs użytkownika.

Podsumowując, jakkolwiek stawiamy poważne wymagania co do kompetencji nauczycieli korzystających z systemu *Daphnis*, dążymy jednak do tego, by korzystanie z bogactwa możliwości systemu było jak najłatwiejsze. Nie oczekujemy jednak, że tworzenie projekcji o charakterze dydaktycznym będzie łatwe, a w szczególności – że da się do końca zautomatyzować.

6.1.4 PROJEKTANT

Spośród różnych grup użytkowników projektanci wyróżniają się tym, że są w równym stopniu obserwatorami, co wizualizatorami. Ich oczekiwania względem systemu sprowadzają się do tego, że pragną oni możliwie niedużym nakładem sił i w możliwie jak najkrótszym czasie uzyskać wizualizację, która pozwoli im na eksplorację realizowanego przez nich oprogramowania. Ich wymagania różnią się w istotny sposób zarówno od wymagań uczniów, jak i nauczycieli.

W zakresie tworzenia własnych wizualizacji, projektanci przede wszystkim spodziewają się, że będzie to łatwe i szybkie. Doświadczenia w zakresie profesjonalnego wykorzystania systemów wizualizacji wskazują, że niespełnienie tego postulatu praktycznie dyskwalifikuje systemy jako użyteczne narzędzia inżynierskie [2]. Należy pamiętać, że w zasadzie powielają one funkcjonalność tradycyjnych, tekstowych debuggerów. Te ostatnie nie dostarczają co prawda informacji w tak atrakcyjny sposób, ale za to natychmiast i bez wysiłku. Nie bez znaczenia są też nawyki i przyzwyczajenia projektantów. Wreszcie nie łatwo jest przekonać zawodowców, że włożenie nieco trudu w przygotowanie wizualizacji może przynieść owoce w postaci uzyskania jakościowo lepszego narzędzia eksploracyjnego, dzięki któremu liczne aspekty trudne lub niemożliwe do wykrycia tradycyjnymi metodami stają się widoczne jak na dłoni.

O środkach ułatwiających przygotowanie wizualizacji wspomnieliśmy już omawiając wymagania stawiane przez nauczycieli. System wizualizacji algorytmów, który miałby być skutecznym narzędziem dla projektantów, musi iść o krok dalej: środki wspomagające pracę wizualizatora muszą zapewnić tworzenie przynajmniej prostych wizualizacji jeśli nie w sposób

w pełni automatyczny, to przynajmniej półautomatyczny. Istniejąca, prototypowa wersja systemu Daphnis nie spełnia jeszcze tego postulatu, niemniej jednak jest bliska jego spełnienia.

Z drugiej strony, projektanci, w porównaniu z innymi grupami użytkowników, stawiają o wiele niższy próg wymagań, jeśli chodzi o repertuar graficzny projekcji. Najważniejsze jest, by była przejrzysta i klarowna, i by stosowane metafory były adekwatne i zrozumiałe. Projektanci nie wymagają dodatkowej zawartości informacyjnej przekazu, poprzestają też na stosunkowo niskim poziomie abstrakcji. Należy pamiętać, że projektant zasadniczo rozumie algorytm będący przedmiotem wizualizacji. Zastosowanie systemu ma na celu weryfikację, na ile faktyczne zachowanie oprogramowania pokrywa się z założeniami i przewidywaniami projektanta.

W zakresie interfejsu użytkownika, podczas obserwacji projekcji, wymagania projektanta nie wykraczają poza wymagania ambitnego ucznia – eksperymentatora.

6.1.5 POTRZEBY I WYMAGANIA UŻYTKOWNIKÓW – PODSUMOWANIE

Prostota obsługi systemu jest jednym z zasadniczych postulatów. Początkujący uczniowie potrzebują łatwego w użyciu interfejsu użytkownika podczas obserwacji projekcji, natomiast projektanci poszukują narzędzia, które umożliwi im wygenerowanie prostych wizualizacji bez większego wysiłku. Szczególnie pożądane by były rozwiązania automatyczne lub półautomatyczne. Im bardziej jednak skomplikowana jest wizualizacja, tym bardziej może być skomplikowany proces jej przygotowania. Pokazy o przeznaczeniu dydaktycznym, charakteryzujące się najwyższym poziomem abstrakcji, mogą wymagać od użytkownika dobrej znajomości systemu i znacznego nakładu pracy. Nawet jednak w tym przypadku nie chcielibyśmy, by reakcją użytkownika było przerażenie, irytacja lub znudzenie.

Jeśli chodzi o zasób środków graficznych systemu, podporządkowany on musi być zastosowaniom dydaktycznym, jako narzucającym najwyższy próg wymogów. Użytkownicy niezwiązani z dydaktyką zadowolą się podzbiorem tych możliwości.

W kolejnych podrozdziałach omówimy:

- interfejs obserwatora, zaprojektowany tak, by był dostatecznie prosty i zrozumiały dla uczniów – widzów,
- zasoby graficzne, umożliwiające tworzenie wizualizacji o istotnych walorach dydaktycznych,
- interfejs wizualizatora, pozwalający w pełni wykorzystać możliwości systemu do tworzenia wizualizacji wysoce abstrakcyjnych,

- środki wspomagające tworzenie wizualizacji, ułatwiające pracę nauczycieli i nadające systemowi charakter narzędzia inżynierskiego.

6.2 INTERFEJS OBSERWATORA W SYSTEMIE *DAPHNIS*

Uruchomienie dowolnego programu, który został uprzednio przygotowany do projekcji (animowany), powoduje automatyczne rozpoczęcie projekcji algorytmu. Wiąże się to z otwarciem okna projekcyjnego, które implementuje wszystkie najważniejsze elementy interfejsu użytkownika dostępne podczas obserwacji projekcji i ewentualnej z nią interakcji. Niezależnie od tego system *Daphnis* zawiera niewielką aplikację ułatwiającą uaktywnienie wskazanej przez użytkownika projekcji. W szczególności dotyczyć to może jednej ze standardowych animacji, rozpowszechnianych wraz z systemem.

6.2.1 ROZPOCZĘCIE PRACY Z SYSTEMEM

Najłatwiejszym sposobem rozpoczęcia pracy z systemem *Daphnis* jest uruchomienie aplikacji o nazwie *DAPHNIS.EXE*. Po prawidłowym zainstalowaniu systemu powinna ona być dostępna z poziomu menu startowego systemu *Windows 9x*.

Aplikacja *DAPHNIS.EXE* została sporządzona w postaci tzw. kreatora (ang. *wizard*). W jej skład wchodzą kolejne strony pomagające w sposób efektywny z niej korzystać.

Strona pierwsza (plansza 6.1) zawiera podstawowe informacje o programie oraz przycisk umożliwiający połączenie z witryną internetową poświęconą wizualizacji oprogramowania oraz systemowi *Daphnis*.

Druga strona kreatora *Daphnis* (plansza 6.2) pozwala na wybranie i uruchomienie jednej ze standardowych projekcji przykładowych. Aplikacja samoczynnie wybiera odpowiedni plik z programem i uruchamia go w powiązaniu z właściwym skryptem konfiguracyjnym. Istnieje też możliwość wskazania własnej projekcji, niekoniecznie należącej do zestawu standardowego. W takim przypadku należy przejść do strony trzeciej.

Strona trzecia (plansza 6.3) pozwala na indywidualne wskazanie przez użytkownika pliku wykonywalnego, zawierającego program mający sterować projekcją, oraz skryptu konfiguracyjnego. Zadanie ułatwiają przyciski powodujące wyświetlenie standardowych okien dialogowych do wyboru pliku (*file dialog*). Dodatkowym ułatwieniem jest fakt, że program przechowuje informacje na temat pewnej liczby ostatnio wywoływanych projekcji i pozwala je wybrać z rozwijalnej listy powiązanej z polem edycyjnym nazwy pliku – zachowując przy tym zapamiętane skojarzenia pro-

gramów z użytymi dla nich skryptami. Użytkownicy, którzy na drugiej stronie kreatora dokonali wyboru jednej ze standardowych wizualizacji, nie mają dostępu do strony trzeciej.

6.2.2 SZYBKIE URUCHAMIANIE PROJEKCJI

Oprócz sposobu opisanego w poprzednim punkcie, możliwe jest też uruchomienie projekcji poprzez bezpośrednie wywołanie pliku wykonywalnego zawierającego program przeznaczony do wizualizacji. Pierwszą operacją wykonywaną przez tak uruchomiony program jest wyświetlenie okienka dialogowego przypominającego trzecią stronę kreatora aplikacji *DAPHNIS.EXE* (plansza 6.4). Umożliwia ono wybór skryptu konfiguracyjnego. Poza standardowym okienkiem dialogu plikowego, możliwe jest też skorzystanie z listy zarejestrowanych nazw poprzednio używanych skryptów. Ostatnia z nich jest domyślnie wpisana w polu edycyjnym, tak więc jej ponowne użycie wymaga jedynie zatwierdzenia dialogu przez użytkownika. Opisywane okienko, będąc stroną konwencjonalnego kreatora, pozwala na przejście do strony zawierającej informacje o programie i wyglądającej tak samo, jak pierwsza strona aplikacji *DAPHNIS.EXE* (plansza 6.1).

Użytkownicy, którzy chcą pominąć etap interaktywnego odpytywania użytkownika, mogą podać ścieżkę dostępu do skryptu konfiguracyjnego w parametrach linii wywołania poddawanego wizualizacji programu, wywołując ten program według następującego wzorca:

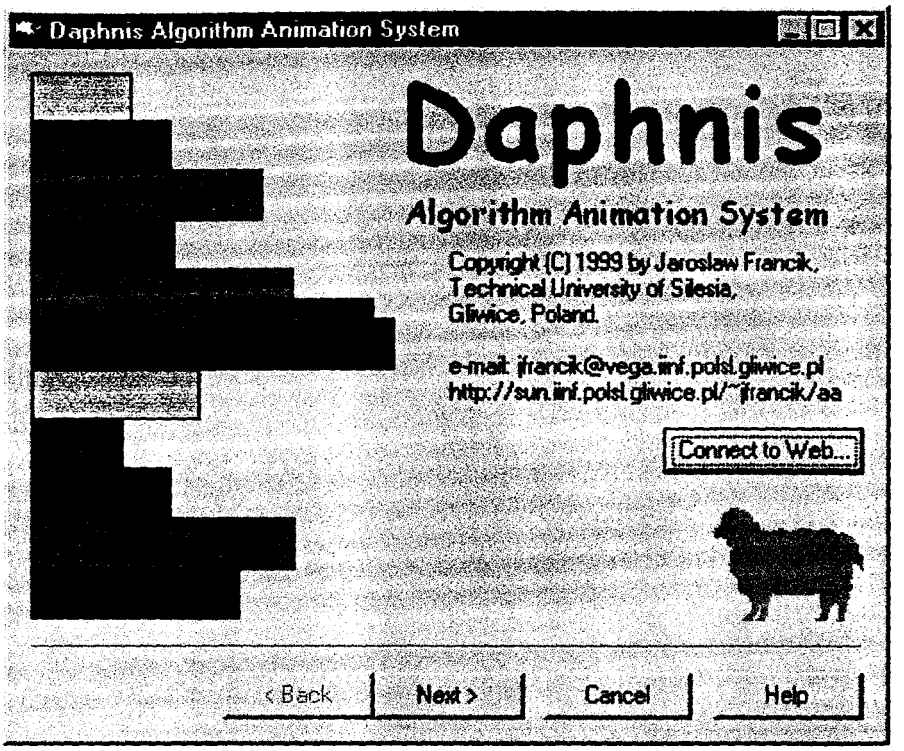
nazwa_programu **-daphnis** *nazwa_skryptu*

6.2.3 STEROWANIE PRZEBIEGIEM PROJEKCJI I OBRAZEM

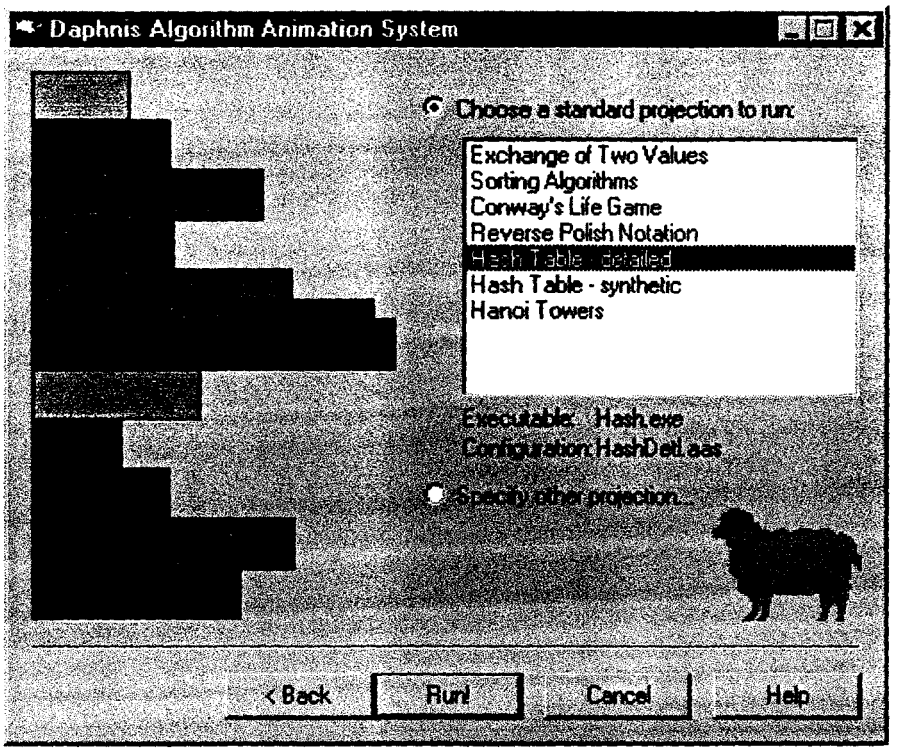
Typowy wygląd okna projekcyjnego przedstawia plansza 6.5. Zawiera ono typowe dla aplikacji systemu *Windows* elementy, takie jak pasek tytułu, menu, pasek narzędzi (ang. *toolbar*), obszar roboczy i pasek statusu. W obszarze roboczym dodatkowo dostępne jest menu kontekstowe (wywoływane za pomocą prawego klawisza myszy), oferujące najważniejsze opcje.

Podstawowe dostępne dla użytkownika polecenia przedstawiają się następująco:

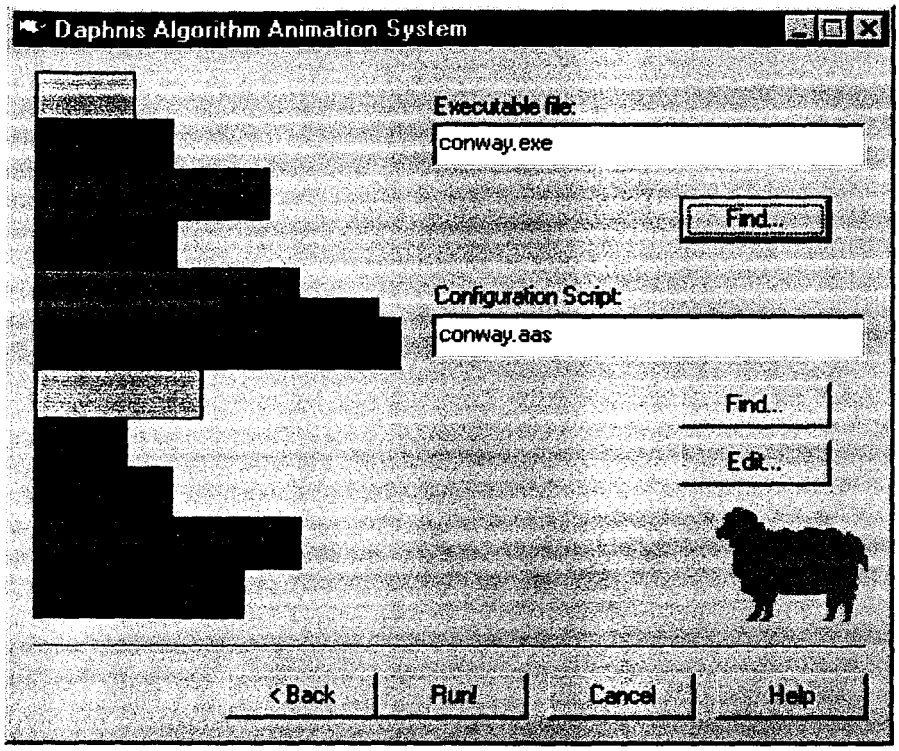
- **Polecenia sterujące przebiegiem projekcji**, to jest powodujące jej rozpoczęcie, zakończenie a także chwilowe przerwanie (pauzę) i wznowienie, są reprezentowane zarówno przyciskami w pasku narzędzi, jak i w głównym oraz kontekstowym menu. W pasku narzędzi odwołujemy się do powszechnie znanej metafory, stosując standardowe oznaczenia używane w sprzęcie audio-video (plansza 6.6).



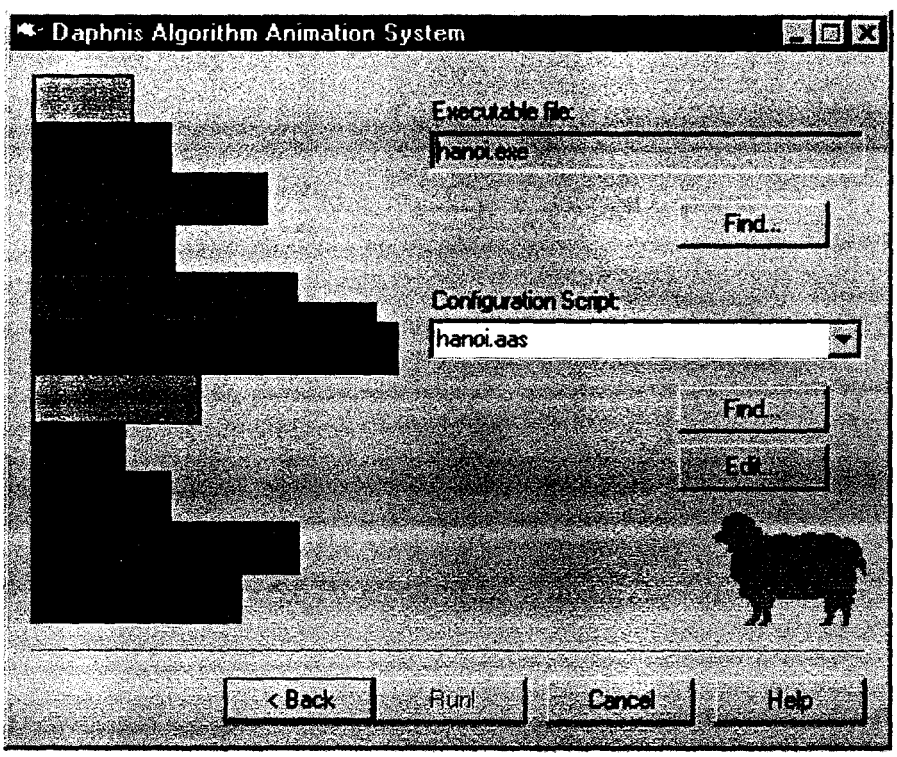
Plansza 6.1 Program *DAPHNIS.EXE*, strona pierwsza – informacje o programie



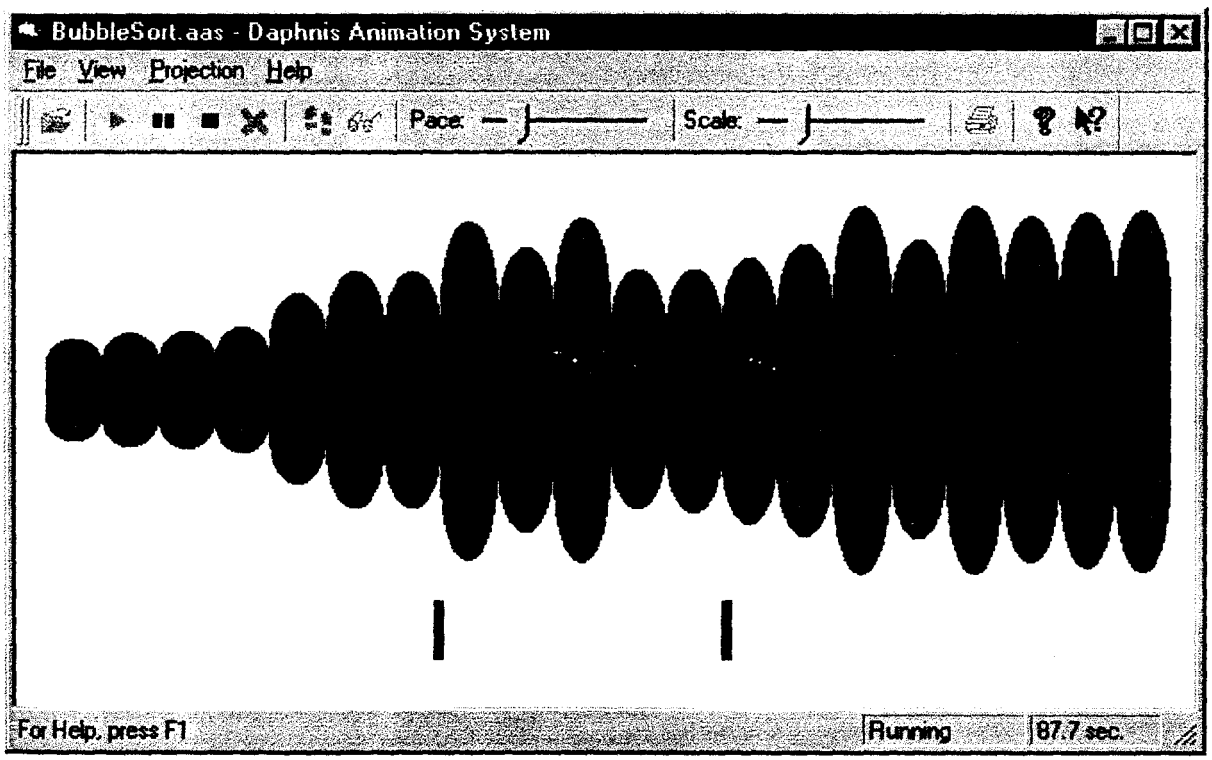
Plansza 6.2 Program *DAPHNIS.EXE*, strona druga
– wybór standardowej projekcji demonstracyjnej



Plansza 6.3 Program *DAPHNIS.EXE*, strona trzecia – wybór programu przeznaczonego do projekcji oraz skryptu konfiguracyjnego

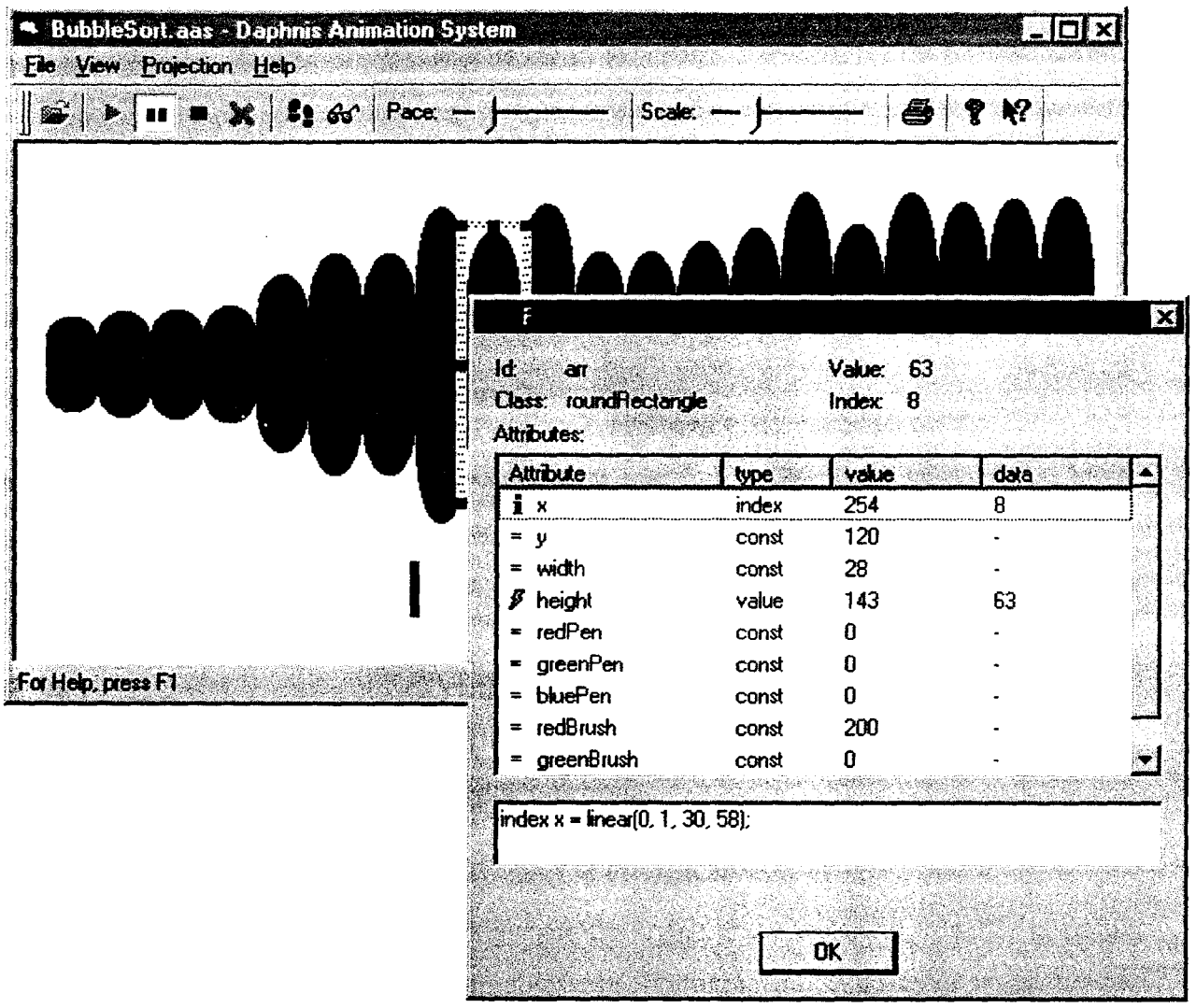


Plansza 6.4 Okienko służące do wyboru skryptu konfiguracyjnego przy szybkim uruchamianiu projekcji



Plansza 6.5 Okno projekcyjne systemu *Daphnis*

Plansza 6.6 Pasek narzędzi systemu *Daphnis*



Plansza 6.7 Okienko dialogowe inspekcji danych
(właściwości elementu graficznego) w systemie *Daphnis*

- **Polecenie pracy w trybie krokowym** jest aktywne tylko po uprzednim przerwaniu projekcji (przejściu w tryb pauzy). Wykonanie polecenia powoduje wykonanie i wizualizację jednostkowej operacji algorytmu (to jest takiej, która jest reprezentowana przez pojedynczą akcję graficzną na ekranie). Podobnie, jak opcje sterowania przebiegiem projekcji, polecenie pracy krokowej jest dostępne zarówno w pasku narzędzi, jak i w obydwu menu.
- **Sterowanie tempem projekcji** jest dostępne wyłącznie z poziomu paska narzędzi. Zawiera on suwak pozwalający na płynne regulowanie tempa. Dodatkowo, w obszarze paska stanu, wyświetlany jest zegar umownego czasu systemowego. Czas ten biegnie szybciej, gdy tempo projekcji jest szybsze, i wolniej, gdy tempo jest wolniejsze, tak że w efekcie wszystkie operacje algorytmu trwają w tej skali czasu jednakowo długo, bez względu na ustawienie tempa.
- **Skalowanie obrazu** jest również dostępne tylko z poziomu paska narzędzi. Zawiera on suwak pozwalający na płynne wyregulowanie wielkości wyświetlanego obrazu.
- **Przewijanie obrazu**, pozwalające (w przypadku dostatecznie dużej skali) na określenie, która część obrazu powinna być widoczna w obrębie okna, jest dostępne za pomocą standardowych pasków przewijania, znajdujących się w prawej oraz w dolnej części obszaru roboczego okna.

Ponadto dostępne są opcje: wyjścia z programu (zamknięcia okna projekcyjnego), określenia widoczności poszczególnych elementów okna i uruchomienia systemu pomocy (ang. *help*). Dodatkowe polecenie inspekcji danych zostanie omówione w następnym punkcie. W tab. 6.2 zgrupowano wszystkie dostępne polecenia.

6.2.4 INTERAKCJA Z WIZUALIZOWANYM PROGRAMEM

Podobnie, jak wykonanie operacji algorytmu w trybie pracy krokowej, również operacje związane z interakcją z wizualizowanym algorytmem (programem) są dostępne wyłącznie po uprzednim przejściu w tryb pauzy. Podstawowym poleceniem jest opcja inspekcji danych, dostępna w menu głównym programu oraz w menu kontekstowym. Użytkownik musi wskazać obiekt graficzny na ekranie, który reprezentuje dane go interesujące. Może się to odbyć poprzez proste wskazanie kursorem myszy i naciśnięcie lewego przycisku. Odpowiedni obiekt zostanie wyróżniony za pomocą ramki (plansza 6.7), a polecenie inspekcji danych w menu głównym odblokuje się. Alternatywną metodą jest wskazanie obiektu kursorem myszy wraz z naciśnięciem prawego klawisza myszy: spowoduje to otwarcie się menu kontekstowego z odblokowaną opcją inspekcji danych. Jeśli żaden element graficzny nie jest wyróżniony, wówczas opcje inspekcji danych pozostają zablokowane.

Menu	Polecenie	Opis	Klawisz	Przycisk
File	Open script...	Pozwala wybrać skrypt konfiguracyjny	Ctrl+O	
	Edit script...	Rozpoczyna edycji skryptu konfiguracyjnego	Ctrl+E	
	Print...	Drukuje aktualny kadr projekcji	Ctrl+P	
	Print preview	Włącza tryb podglądu wydruku		
	Print setup	Pozwala ustawić parametry drukarki		
	Exit	Kończy pracę programu	Alt + F4	
View	Toolbar	Przełącza widoczność paska narzędzi		
	Status bar	Przełącza widoczność paska stanu		
Projection	Start	Rozpoczyna projekcję	F5	
	Pause/Resume	Zatrzymuje lub wznawia projekcję	F6	
	Stop	Kończy projekcję	F7	
	Single step	Animuje operację w trybie krokowym	F2	
	Inspect	Wyświetla okno inspekcji danych	F4	
	Clear	Usuwa elementy graficzne z okna projekcji		
Help	Help	Wyświetla okno pomocy	F1	
	Index	Wyświetla spis treści pomocy		
	About Daphnis	Wyświetla informacje o programie		
		Określa tempo projekcji	Ctrl ←→	
		Określa skalowanie obrazu	Ctrl ↑↓	

Tab. 6.2. Zestawienie poleceń dostępnych użytkownikowi podczas obserwacji projekcji

Wywołanie polecenia inspekcji danych powoduje wyświetlenie okna inspekcji danych (plansza 6.7) dla wskazanego elementu graficznego. Sposób wyświetlania elementu, to jest jego atrybuty takie, jak położenie, rozmiar, kolor itp., są uzależnione od określonych wielkości pobranych z wnętrza wizualizowanego programu. Mogą to być wartości zmiennych lub indeksy elementów tablic. Okno inspekcji danych wyświetla listę tak powiązanych atrybutów elementu graficznego, wraz z aktualnymi wartościami danych pobranymi z programu. W ten sposób użytkownik otrzymuje zapis wizualizowanych wielkości w postaci symbolicznej. Co więcej, niektóre z tych wartości może też modyfikować: po zatwierdzeniu takiej zmiany (za pomocą przycisku OK) struktury danych wizualizowanego programu są odpowiednio modyfikowane, a wyświetlany ich obraz graficzny stosowanie uaktualniany.

W jednej z poprzednich wersji systemu, noszących jeszcze nazwę WinSANAL, dostępna była również opcja interakcji z wizualizowanym programem metodą bezpośredniej manipulacji [109]. Opcja ta została zaimplementowana w głównym stopniu dzięki udziałowi w pracach M. Nowaka [111]. W rozdziale 9.7 przedstawiamy wizualizację stworzoną za pomocą tej wersji systemu, przedstawiającą symulację systemu ewolucyjnego.

Zagadnieniom bezpośredniej manipulacji obiektami poświęcono wiele publikacji [34, 35, 41, 79, 101, 102], a technologia ta jest obecnie już *de facto* standardem przemysłowym. W omawianej wersji systemu, po zaznaczeniu elementu graficznego na ekranie wyświetlana była obwódka z zaznaczonymi miejscami przeznaczonymi do przeciągania elementu. W ten sposób można było zmienić zarówno jego położenie, jak i rozmiary. Wprowadzone zmiany podlegały interpretacji i były tłumaczone na odpowiednią modyfikację wartości struktur danych. I tak, jeśli na przykład określona zmienna była reprezentowana w postaci prostokąta, którego wysokość odzwierciedlała wartość tej zmiennej, wówczas zmieniając wysokość prostokąta można było bezpośrednio manipulować wartością zmiennej. Ciekawe rozwiązanie zastosowano przy wizualizacji tablic w postaci ciągu elementów graficznych leżących jeden obok drugiego. Przesunięcie elementu w taki sposób, że zmieniało to sekwencję ułożenia elementów na ekranie, powodowało analogiczną operację wykonaną na elementach tablicy, przy czym użytkownik miał wybór pomiędzy trzema możliwymi interpretacjami:

- Wartość była kopiowana w nowe miejsce. Poprzednia wartość elementu na tym miejscu ulegała zatarciu.
- Przesuwany element był zamieniany z elementem docelowym.
- Wartość była przesuwana na nowe miejsce, a elementy tablicy znajdujące się pomiędzy jego poprzednią a nową pozycją były przesuwane o jedną pozycję w lewo lub w prawo tak, by żaden z elementów tablicy nie uległ zatarciu.

W toku prac implementacyjnych nad nowym systemem *Daphnis* opisana powyżej funkcja została przejściowo usunięta.

6.3 REPERTUAR ŚRODKÓW GRAFICZNYCH

Z rozważań przedstawionych w punkcie 6.1 wynika, że repertuar dostępnych w systemie *Daphnis* środków graficznych został wyznaczony przede wszystkim przez zastosowania dydaktyczne tego systemu. Za krytyczne uznaliśmy następujące główne czynniki:

- Zrozumiałość i klarowność,

- Zawartość informacyjną i wysoki poziom abstrakcji,
- Atrakcyjność formy.

W kolejnych podpunktach scharakteryzujemy repertuar środków graficznych oferowanych przez *Dahnis* w odniesieniu do wyżej wymienionych czynników. W tym miejscu przedstawimy opis na poziomie stosunkowo ogólnym, więcej szczegółów można znaleźć w następnym punkcie (6.4), poświęconym interfejsowi wizualizatora.

6.3.1 ZROZUMIAŁOŚĆ I KLAROWNOŚĆ

Ofertę graficzną systemu *Daphnis* można najkrócej określić jako *płynną animację sterowaną przepływem danych*. Wizualizację opartą na takiej zasadzie przeanalizowaliśmy w rozdziale 4, dowodząc, że ułatwia ona zrozumienie algorytmu przez człowieka.

Za to, by wizualizacje były zrozumiałe, a użyte środki graficzne – metafory rzeczywistych zdarzeń – trafne, odpowiada w głównej mierze projektant wizualizacji. System animacji musi natomiast odznaczać się na tyle bogatą i elastyczną ofertą graficzną, by zapewnić mu odpowiedni poziom swobody twórczej. Proces projektowania repertuaru graficznego *Daphnis* był przede wszystkim zdeterminowany przez trzy czynniki. Oto one:

- **Różnorodność** dostępnych elementów graficznych. Obecna wersja systemu zawiera większą część środków graficznych dostępnych w innych porównywalnych systemach. Oferta ta jest ciągle poszerzana i będzie wzbogacana w przyszłości. W tej chwili obejmuje elementy graficzne rozmaitych kształtów, w tym przystosowane do wyświetlania danych w postaci symbolicznej, elementy definiowane za pomocą rastrowych obrazów graficznych, a także specjalny element przeznaczony do udźwiękowienia (ang. *auralisation*) projekcji, czyli wyrażenia pewnych treści za pomocą dźwięków. Za istotny mankament w oferowanym repertuarze należy uznać brak środków skutecznie reprezentujących struktury dynamiczne takie, jak grafy i drzewa. Jest to zagadnienie tyleż interesujące, co obszerne [30], i wykracza poza ramy niniejszej pracy. Badania w tym kierunku prowadził w jednej z poprzednich wersji systemu M. Nowak [109, 111], a obecnie problematyką tą zajmuje się K. Dobosz [29], który pracuje nad wizualizacją algorytmów symulujących systemy ewolucyjne [196].
- **Elastyczność**. Wyraża się ona pełną swobodą łączenia rozmaitych elementów konstrukcyjnych wizualizacji. Dane pobrane z programu mogą być przekształcone praktycznie bio-

rąc w dowolny sposób tak, by kontrolować różnorodne atrybuty elementów graficznych. Problematykę tę opiszemy dokładniej w punkcie 6.4.

- **Otwartość** systemu. Zasób oferowanych środków graficznych może być poszerzany, poprzez dołączanie do systemu kolejnych modułów *ActiveX*. W istocie implementacja systemu zaczęła się od stworzenia zupełnie elementarnego zestawu środków, które były sukcesywnie poszerzane w miarę tworzenia kolejnych projekcji. Mechanizm ten będzie w przyszłości powodował, że repertuar graficzny systemu *Daphnis* będzie ustawicznie wzbogacany o nowe środki.

6.3.2 ZAWARTOŚĆ INFORMACYJNA I POZIOM ABSTRAKCJI

Wysoki poziom abstrakcji niemal automatycznie zwiększa zawartość informacyjną pokazu, gdyż nieodłącznie wiąże się z wprowadzeniem tzw. zawartości intencyjnej (por. rozdział 1.2.2). Dzięki zasygnalizowanej w poprzednim punkcie elastyczności łączenia elementów systemu i wiązania ich z danymi pobieranymi z wizualizowanego programu, twórca wizualizacji może tworzyć rozmaite, wysoce przetworzone reprezentacje poddawanego wizualizacji algorytmu. Możliwe jest wykrywanie określonych stanów, w jakich znajduje się proces wykonania algorytmu, dekodowanie istotnych operacji wykonywanych w danej chwili, zbieranie danych pozwalających na obliczenie różnorodnych wyników analitycznych czy utworzenie historii procesu wykonania algorytmu. Wszystkie zebrane w ten sposób informacje mogą zostać zobrazowane w dogodnej formie graficznej.

Osobnym zagadnieniem jest wprowadzenie do projekcji elementów, które w żaden sposób (nawet po przetworzeniu) nie mogą być uzależnione od danych występujących w obrębie poddawanego wizualizacji programu. System *Daphnis* oferuje zestaw elementów graficznych niezależnych – z technicznego punktu widzenia – od kontekstu obserwowanego programu. Są to:

- Podpisy wyjaśniające znaczenie poszczególnych elementów obrazu lub komentujące cały widoczny obraz, na przykład wyjaśniające, w której fazie wykonania znajduje się w danej chwili algorytm.
- Elementy tła, takie jak osie liczbowe z podziałkami ułatwiające szacowanie wartości danych, elementy porządkujące obiekty graficzne na ekranie (na przykład pudełko zawierające elementy tablicy czy pałeczka, na którą nanizano krążki wieży Hanoi), czy wreszcie tory wyznaczające typowe kierunki przepływu danych.

6.3.3 ATRAKCYJNOŚĆ FORMY

Atrakcyjność formy nie tylko zwiększa zainteresowanie i zaangażowanie obserwatora projekcji, ale także ułatwia dostrzeżenie pewnych niuansów, które w mniej atrakcyjnej szacie dałyby się również zauważyć, ale najczęściej uchodziłyby uwagi obserwatora. Tak więc atrakcyjna szata graficzna projekcji ma swój wpływ na podniesienie poziomu zrozumiałości i zwiększa realną zawartość informacyjną wizualizacji. Oczywiście, projektant musi mieć na uwadze, iż nadmiar efektów graficznych może doprowadzić do skutków wręcz odwrotnych – które to zjawisko często określa się jako *efekt Las Vegas* [192].

Przy obecnym rozwoju komputerowych technik graficznych można stosunkowo tanim kosztem uzyskać bardzo ciekawe efekty. Oto główne z nich:

- **Różnorodność** reprezentacji graficznych. Problematykę tę poruszyliśmy omawiając zrozumiałość prezentacji.
- **Płynna animacja**. Wszelkie zmiany stanu elementów graficznych powinny dokonywać się za pomocą płynnie animowanych ruchów, podobnie jak w filmie animowanym. Nagłe, skokowe zmiany są trudniejsze do zauważenia. Stosując animację można łatwiej ukazać związki przyczynowo-skutkowe – jak to ma miejsce przy wizualizacji elementarnych operacji przepływu danych.
- **Kolor**. Umiejętne operowanie kolorem jest niezmiernie istotne we wszystkich dziedzinach projektowania graficznego. W wizualizacji algorytmów kolory mogą być zastosowane do kodowania stanów, w jakich się znajduje algorytm lub jego struktury danych, wyróżniania w obrębie obrazu miejsc interesujących, w których dzieje się coś ważnego, kojarzenia oddalonych od siebie elementów obrazu itp. W systemie *Daphnis* kolor jest traktowany jak jeden z wielu atrybutów elementów graficznych, i można go dowolnie wiązać z atrybutami poddawanego wizualizacji algorytmu. W szczególności, *Daphnis* pozwala wyrażać za pomocą koloru wartość zmiennych – jest to zastosowanie koloru z jakim nie spotkaliśmy się w innych systemach animacji algorytmów.
- **Dźwięk**. Jest to wciąż jeszcze stosunkowo nowe pole eksploracji w dziedzinie wizualizacji algorytmów. *Daphnis* oferuje obiekt pseudo-graficzny, posiadający zerową reprezentację graficzną, wyposażony natomiast w atrybuty dźwiękowe, takie jak wysokość i natężenie dźwięku. Podobnie, jak to ma miejsce w przypadku elementów graficznych, atrybuty te mogą być w dowolny sposób wiązane z atrybutami programu.

- **Estetyka**. Zadziwiające, jak mało miejsca w literaturze dotyczącej wizualizacji algorytmów poświęca się kwestiom estetycznym. Nawet w artykule Browna i Hershbergera o kolorze i dźwięku w animacji algorytmów [19], oba tytułowe środki wyrazu potraktowano czysto funkcjonalnie, nie zastanawiając się nad ich aspektami estetycznymi. Widocznie w 1971 roku W.H. Huggins [201] miał rację nisko oceniając zdolności informatyków do wizualnego wyrażania się (por. strona 40). Również i my jesteśmy zmuszeni stwierdzić, iż to, czy stworzone przy pomocy *Daphnis* projekcje algorytmów będą estetyczne, czy nie, zależy od osób przygotowujących takie wizualizacje, a w znacznie mniejszym stopniu od jakości środków znajdujących się w ofercie systemu.

6.4 INTERFEJS WIZUALIZATORA

Do uruchomienia projekcji w systemie *Daphnis* konieczne jest dostarczenie odpowiednio zmodyfikowanej, wykonywalnej wersji poddawanego wizualizacji programu oraz skryptu konfiguracyjnego. Jak już wspominaliśmy, projektowanie wizualizacji obejmuje następujące trzy główne czynności:

- Uzupełnienie tekstu źródłowego programu,
- Kompilacja i scalenie (linkowanie) programu,
- Stworzenie skryptu konfiguracyjnego.

Dwie pierwsze czynności dają w efekcie wykonywalną wersję programu, przygotowaną do sterowania projekcją. Uruchomienie takiego programu automatycznie inicjuje przeprowadzenie projekcji. Przygotowany program może być uruchamiany w połączeniu z różnymi skryptami konfiguracyjnymi, generując jakościowo różne wizualizacje.

O ile uzupełnienie tekstu źródłowego i kompilacja oraz linkowanie programu przebiegają według dość sztywnych reguł (i dają się zautomatyzować – por. rozdz. 6.5), o tyle przygotowanie skryptu jest pracą twórczą i stanowi najważniejszy etap projektowania wizualizacji. Użytkownicy tworzący zaawansowane wizualizacje iteracyjnie udoskonalają kolejne wersje skryptu. Do stworzenia skryptu konfiguracyjnego niezbędna jest ogólna wiedza na temat sposobu konstruowania wizualizacji w systemie *Daphnis*.

W kolejnych punktach omówimy: uzupełnianie tekstu źródłowego programów, ich kompilację i scalanie, następnie opiszemy sposób tworzenia wizualizacji i zasadę działania projekcji, by wreszcie omówić format skryptu konfiguracyjnego. Na zakończenie podamy kilka uwag dotyczących projektowania bardziej zaawansowanych wizualizacji. Czytelnik zaintere-

sowany dalszymi szczegółami dotyczącymi zasad konstrukcji wizualizacji, sposobu działania systemu i jego architektury, znajdzie te informacje w rozdziale 7.

Wymagania stawiane przez użytkowników – projektantów wymuszają daleko idące uproszczenia interfejsu wizualizatora. System *Daphnis* wychodzi tym potrzebom naprzeciw oferując narzędzia wspomagające pracę wizualizatora. Opisane zostaną one w punkcie 6.5.

6.4.1 UZUPEŁNIENIA TEKSTU ŹRÓDŁOWEGO PROGRAMU

Aby przygotować program do wizualizacji, należy go uzupełnić o wywołania funkcji realizujących operacje maszyny projekcyjnej systemu *Daphnis*. Obecna wersja systemu zawiera interfejs programowy udostępniający te funkcje dla programów napisanych w języku *C* lub *C++*, tym samym otwierając możliwość wizualizacji algorytmów zapisanych w tych właśnie językach. Ponieważ z technicznego punktu widzenia implementacja takiego interfejsu jest bardzo prosta, sprowadza się bowiem do udostępnienia programowi dynamicznej biblioteki typu *dll* (ang. *dynamic link library*), dostosowanie systemu do obsługi dowolnego języka programowania nie jest problemem. Jak dotąd, próbę taką – poza *C* i *C++* – podjęto dla języka *Visual Basic* [206, 207].

Aby uniknąć ewentualnych konfliktów nazw, definicje wszystkich funkcji realizujących operacje elementarne zostały zadeklarowane w oddzielnej przestrzeni nazewniczej (ang. *namespace*), o nazwie *daphnis*. Tak więc użytkownik powinien zadeklarować użycie tego obszaru za pomocą dyrektywy *using namespace* albo każdorazowo stosować identyfikatory kwalifikowane. Rozwiązanie to jest w łatwy sposób dostępne jedynie dla programów napisanych w języku *C++*. Na użytek innych języków (w tym *C*), ze względu na skomplikowane reguły łączenia z funkcjami zadeklarowanymi w *C++*, zdecydowaliśmy się na zdublowanie całego interfejsu z wykorzystaniem funkcji nazywanych zgodnie z konwencją języka *C* (to znaczy *extern "C"*). Pociągnęło to oczywiście konieczność rezygnacji z wykorzystania przestrzeni nazewniczej, a także funkcji przeciążonych. Nazwy funkcji uległy zmianom poprzez wprowadzenie stałego przedrostka – *daphnis* – oraz przyrostka umożliwiającego rozróżnienie funkcji pierwotnie zadeklarowanych jako przeciążone. Deklaracje wszystkich funkcji umieszczono w pliku nagłówkowym *daphnis.h*. Jest on uniwersalny – może być stosowany zarówno w *C++*, jak i w *C*.

Funkcje, o których wywołania uzupełnia się tekst źródłowy programu poddawanego wizualizacji, można podzielić na dwie grupy. Są to funkcje sterujące projekcją oraz funkcje techniczne (reżyserskie). Te ostatnie stosowane są opcjonalnie.

FUNKCJE STERUJĄCE PROJEKCJĄ

W rozdziale 4.4 zdefiniowane zostały trzy operacje stosowane w algorytmie wizualizacji sterowanej przepływem danych. Dostarczają one systemowi wszystkich niezbędnych informacji na temat sposobu działania poddawanego wizualizacji programu, i są jedynymi funkcjami, które są niezbędne w poddawanym wizualizacji programie. Operacje te to:

- Operacja ***Register***. Zgłasza systemowi pojawienie się nowej zmiennej, która może być ewentualnym przedmiotem wizualizacji. Szczególny przypadek stanowi zgłoszenie zmiennych tablicowych, dla których przewidziano osobną, przeciążoną formę wywołania. Operację zaimplementowano w postaci rodziny przeciążonych funkcji[7], które odpowiadają następującym deklaracjom szablonów (druga deklaracja to forma przeznaczona dla zmiennych tablicowych):

```
template<class T> BOOL Register(char *pId, T *pVar);
template<class T> BOOL Register(char *pId, T *pVar, int nSize);
```

gdzie:

`pId` – symboliczna nazwa zmiennej, umożliwiająca skojarzenie tej zmiennej z odpowiednim zapisem w skrypcie konfiguracyjnym. Z technicznego punktu widzenia może to być dowolna nazwa, zaleca się jednak użycie nazwy identycznej z jej identyfikatorem, w miarę potrzeby kwalifikowanej nazwą funkcji i numerem bloku wewnątrz funkcji.

`pVar` – adres zmiennej.

`nSize` – rozmiar tablicy; występuje tylko w przeciążonej wersji funkcji przeznaczonej do rejestracji zmiennych tablicowych.

Wywołanie operacji *Register* powinno wystąpić w tekście programu po wystąpieniu definicji zmiennej, a wykonanie tej operacji powinno nastąpić przed pierwszym użyciem tej zmiennej.

- Operacja ***Play***. Informuje ona system o wystąpieniu elementarnej zmiany danych (ściślej: elementarnej operacji przepływu danych). Stanowi więc rodzaj opisu operacji wykonywa-

[7] W przypadku implementacji dla języków nie pozwalających na przeciążanie funkcji (np. *C*), poszczególne formy operacji *Register* są obsługiwane przez funkcje o rozróżnialnych nazwach, takich, jak *daphnisRegisterInt* lub *daphnisRegisterIntArray*.

nej przez algorytm. W zależności od liczby zmiennych zaangażowanych w operacji, przewidziano kilka przeciążonych form deklaracji[8]:

```
void Play(void *pTo);
void Play(void *pTo, void *pFrom);
void Play(void *pTo, void *pFrom1, void *pFrom2)
void Play(int nFrom, void *pTo, ...);
void Play(void *pTo, int nArrSize); // forma jednocześnie wizualizu-
                                    // jąca wszystkie elementy tablicy
```

gdzie:

`pTo` – adres zmiennej docelowej, do której wpływają dane.

`pFrom, pFrom1 ... pFrom3` – lista adresów zmiennych źródłowych, z których wypływają dane.

`nFrom` – w wersji funkcji z otwartą listą parametrów, określa liczbę zmiennych źródłowych. Odpowiedniej długości sekwencja wskazań musi nastąpić po tym parametrze.

`nArraySize` – rozmiar tablicy.

Wykonanie operacji *Play* powinno następować po wykonaniu instrukcji przez nią opisywanej. Jeśli operacja ta ma charakter przypisania, wówczas parametr *pTo* odpowiada zmiennej znajdującej się po lewej stronie znaku przypisania, a parametry *pFrom* odpowiadają zmiennym znajdującym się po prawej stronie. Zaleca się – w miarę możliwości – podawać te zmienne w kolejności od najważniejszej do najmniej ważnej – w subiektywnym odczuciu projektanta. Pierwsza z podanych form, jednoargumentowa, przeznaczona jest do opisu operacji, w których nie występują zmienne będące źródłami danych, na przykład przy przypisaniach *a priori* w rodzaju $a = 20$.

- Operacja ***UnRegister***. Wyrejestrowuje ona wskazaną zmienną, to znaczy informuje system o jej zaniku i powoduje zaprzestanie dalszej wizualizacji tej zmiennej. Operacja ta ma tylko jedną formę:

```
void UnRegister(char *pId);
```

gdzie:

`pId` – adres zmiennej przeznaczonej do wyrejestrowania.

[8] W przypadku implementacji dla języków nie pozwalających na przeciążanie funkcji (np. *C*), poszczególne formy operacji *Play* są obsługiwane przez funkcje o rozróżnialnych nazwach, takich, jak *daphnisPlay1*, *daphnisPlay2* itd.

Operacja *UnRegister* powinna następować w tekście programu po ostatnim użyciu zmiennej, przed jej dealokacją. Należy jednak zwrócić uwagę, iż wyrejestrowanie wszystkich zmiennych biorących udział w projekcji spowoduje usunięcie wszystkich elementów graficznych z okna projekcji, co często nie jest pożądanym stanem końcowym projekcji. Aby tego uniknąć, można zastosować jedną z przedstawianych dalej funkcji technicznych (*Stop*) lub też nie stosować funkcji *UnRegister* w odniesieniu do wybranych zmiennych.

- Funkcja ***Flush***. Stanowi poszerzenie zaproponowanego w rozdziale 4.4 zestawu operacji. Wywołanie funkcji przerywa zbieranie przez system informacji o operacjach, które mogą być zobrazowane jako równoczesne, i powoduje natychmiastowe przystąpienie systemu do wizualizacji wszystkich zarejestrowanych dotąd operacji algorytmu, których przedstawianie jeszcze się nie zaczęło.

Przykładowy tekst źródłowy programu poszerzony o wywołania przedstawionych powyżej funkcji zamieszczono na stronie 79.

W zasadzie wystarczające jest wprowadzanie adnotacji dotyczących tylko tych zmiennych, które mają być uwidocznione podczas wizualizacji. Zaleca się jednak nieco szersze ujęcie, zawczasu uwzględniające również i te zmienne, których uwidocznienia początkowo nie planowano. W razie włączenia ich w późniejszym czasie do wizualizacji można zaoszczędzić na dodatkowym wysiłku polegającym na powtórnej modyfikacji tekstu źródłowego oraz rekompilacji. W skrajnym przypadku można opisać wszystkie zmienne: wprowadza to jedynie niewielki dodatkowy narzut czasowy.

FUNKCJE TECHNICZNE (REŻYSERSKIE)

Funkcje techniczne pozwalają użytkownikowi sterować przebiegiem projekcji i jej parametrami.

Do najczęściej wykorzystywanych funkcji technicznych należą funkcje ***Reset*** i ***Run***. Ich prawidłowe użycie jest warunkiem pełnego wykorzystania interfejsu użytkownika, gdyż pozwalają tak zorganizować projekcję, by możliwe było jej wielokrotne przerywanie i powtarzanie za pomocą przycisków *Start* i *Stop* dostępnych na poziomie okna projekcji. Jest to nieodzowny element projekcji o przeznaczeniu demonstracyjnym lub dydaktycznym. Program nie korzystający z żadnej ze wspomnianych funkcji może tylko jednokrotnie sterować projekcją. Aby powtórzyć projekcję, należy zakończyć taki program i uruchomić go ponownie.

Jednym ze sposobów zorganizowania projekcji w taki sposób, by możliwe było wielokrotne jej powtarzanie, jest cykliczne wykonywanie fragmentu programu implementującego wizualizowany algorytm. Każdorazowe wykonanie tego fragmentu powinno być poprzedzone wywołaniem funkcji ***Reset***, która aktywizuje opcję *Start* dostępną z poziomu okna projekcyjnego. Wywołanie to jest też niezbędne do wznowienia projekcji w przypadku, gdy uprzednio zastosowano opcję *Stop*.

Alternatywnym sposobem zorganizowania projekcji jest stworzenie funkcji stanowiącej punkt wejściowy fragmentu programu implementującego wizualizowany algorytm. Wskaźnik na taką funkcję należy następnie przekazać systemowi jako parametr funkcji ***Run***, która zapewnia właściwą obsługę opcji *Start* i *Stop*.

Inną często stosowaną funkcją techniczną jest ***Stop***. Wprawdzie powiela ona działanie opcji o tej samej nazwie, dostępnej na poziomie okna projekcji, ale równocześnie umożliwia jawne zdefiniowanie miejsca, w którym system przestaje śledzić zachowanie poddawanego wizualizacji oprogramowania. W szczególności można w ten sposób uniknąć zmazania z ekranu reprezentacji graficznej zmiennych, które zostają wyrejestrowane.

Pozostałe funkcje techniczne umożliwiają programowe wywoływanie opcji normalnie dostępnych na poziomie okna projekcji, takich, jak pauza czy sterowanie tempem projekcji.

Poniżej zestawiamy nagłówki wszystkich funkcji technicznych dostępnych w systemie *Daphnis*:

```
void Reset();
void Run(void(*f)());
void Start();           // rozpoczyna projekcję
void Pause();           // załącza lub wyłącza tryb pauzy
void Stop();            // zatrzymuje projekcję
void Clear();           // usuwa z ekranu wszystkie elementy graficzne
void SetPace(int nPace);      // umożliwia sterowanie tempem projekcji
int GetPace();
void SetScale(int nPace);     // umożliwia sterowanie skalą obrazu
int GetScale();
BOOL LoadConfiguration(char *pScriptFileName);
     // pozwala na wczytanie i interpretację skryptu konfiguracyjnego
```

6.4.2 KOMPILACJA I SCALANIE PROGRAMÓW PRZEZNACZONYCH DO WIZUALIZACJI

Po wprowadzeniu modyfikacji do tekstu źródłowego programu poddawanego wizualizacji, jego kompilacja i scalanie nie stanowi problemu. Jak wspomniano wyżej, deklaracje wszystkich wprowadzanych do tekstu źródłowego funkcji znajdują się w pliku nagłówkowym *daphnis.h*. Odpowiednio, plik z biblioteką przeznaczoną do statycznego scalenia z progra

mem nosi nazwę *daphnis.lib*, a odpowiedni plik biblioteki dynamicznej – *daphnis.dll*. W przypadku korzystania z kompilatora *Microsoft Visual C++* [72], plik *daphnis.lib* należy włączyć do bieżącego projektu. Plik *daphnis.dll* jest umieszczany podczas instalacji systemu w folderze systemowym *Windows*, a zatem jest dostępny dla wszystkich programów.

6.4.3 GRELE, ICH KLASY I ATRYBUTY

Gdy dysponujemy już programem przygotowanym do wizualizacji, do otrzymania projekcji brakuje tylko skryptu konfiguracyjnego. Do jego stworzenia niezbędne są elementarne informacje dotyczące struktury wizualizacji w systemie *Daphnis*.

Podstawowym elementem, z którego złożone są obrazy graficzne *Daphnis*, jest **grel** (ang. *GRaphical ELement*) [109, 111]. Chociaż nazwa ta nie występuje w składni skryptów konfiguracyjnych, a zatem nie musi koniecznie być znana wizualizatorowi, sądzimy jednak, że jej użycie poprawi zrozumiałość tekstu.

Grel jest obiektem – w sensie technik obiektowych. Wyróżniamy zatem **klasy greli**, a grele należące do poszczególnych klas charakteryzują się określonym zbiorem atrybutów. Istnieje też hierarchia klas greli, co oznacza, że pewne klasy dziedziczą atrybuty innych klas.

Nie istnieje atrybut, który byłby właściwy dla wszystkich możliwych greli. W terminach techniki obiektowej powiedzielibyśmy, że nadrzędna klasa greli (*Grel*) jest pozbawiona atrybutów. Jest to też klasa abstrakcyjna, to znaczy, że w wizualizacji nie mogą brać udziału grele klasy *Grel*. Zdefiniowano więcej takich abstrakcyjnych klas. Najważniejsze z nich to *XYGrel* i *SolidGrel*. Pierwsza z nich grupuje obiekty posiadające określone rozmiary i pozycję na ekranie. Druga klasa pochodzi od pierwszej, a oprócz rozmiarów i pozycji zdefiniowano w jej ramach również i kolorystykę. I tak, w ramach klasy *XYGrel*, zdefiniowano następujące atrybuty:

- *x* – określa odciętą położenia elementu na ekranie,
- *y* – określa rzędną położenia elementu na ekranie,
- *width* – określa poziomy rozmiar elementu (szerokość),
- *height* – określa pionowy rozmiar elementu (wysokość).

W ramach klasy *SolidGrel*, zdefiniowano następujące dodatkowe atrybuty:

- *redPen* – określa udział barwy czerwonej w kolorze obwódki elementu,
- *greenPen* – określa udział barwy zielonej w kolorze obwódki elementu,
- *bluePen* – określa udział barwy niebieskiej w kolorze obwódki elementu,
- *redBrush* – określa udział barwy czerwonej w kolorze wypełnienia elementu,

- *greenBrush* – określa udział barwy zielonej w kolorze wypełnienia elementu,
- *blueBrush* – określa udział barwy niebieskiej w kolorze wypełnienia elementu.

Wszystkie powyższe kolory określa się zgodnie z regułami systemu RGB, ze współczynnikami określającymi udział poszczególnych barw podstawowych mieszczącymi się w zakresie 0..255.

Jeszcze raz zauważmy, że zarówno *XYGrel*, jak i *SolidGrel* są klasami abstrakcyjnymi, i jako takie nie mogą być bezpośrednio zastosowane w ramach wizualizacji.

Przykładami klas greli, które mogą być zastosowane w wizualizacji, są klasy: *rectangle*, *roundRect* i *ellipse*, grupujące odpowiednio grele w kształcie prostokątów, prostokątów z zaokrąglonymi rogami i elips. Dla wszystkich określone jest położenie na ekranie, rozmiary tudzież kolor obwódki i wypełnienia – ich klasy są bowiem wyprowadzone z klasy *SolidGrel*. Inna klasa greli, *bitmap*, jest wyprowadzona z klasy *XYGrel* – zatem dla jej elementów zdefiniowane jest położenie na ekranie i rozmiary, ale nie kolory. Zamiast tego obiekty tej klasy są powoływane do życia ze wskazaniem na określony plik grafiki rastrowej, który definiuje ich wygląd: kształt i kolorystykę. Inna klasa greli, *string*, została wyprowadzona z klasy *rectangle* i uzupełniona o nowy atrybut: *text*. Dzięki temu obiekty tej klasy wyświetlają łańcuchy tekstowe wewnątrz kolorowych prostokątów.

Należy zaznaczyć, że otwarta architektura systemu *Daphnis* pozwala dynamicznie wzbogacać go o nowe klasy greli, gotowe do wykorzystania w nowych wizualizacjach. Wymaga to oczywiście znacznie głębszej znajomości architektury systemu, a przede wszystkim interfejsów klas greli; problemy te omawiamy w rozdziale 7.

6.4.4 TWORZENIE GRAFICZNYCH REPREZENTACJI DANYCH

Dane występujące w poddawanym wizualizacji programie mogą po przetworzeniu sterować wartościami odpowiednich atrybutów greli. W procesie tym bierze udział dodatkowy element tłumaczący, zwany **translatorem**, którego zadaniem jest takie przetworzenie danych pobranych z programu, by w odpowiedni sposób mogły sterować sposobem wyświetlania elementów graficznych. Grel, którego atrybuty powiązano w ten sposób z określonym podzbiorem danych występujących w poddawanym wizualizacji programie, staje się **graficzną reprezentacją** tychże danych.

Dla przykładu, jeśli graficzną reprezentacją wartości pewnej zmiennej w programie ma być grel w kształcie prostokąta, wówczas możemy powiązać wybrany atrybut (lub atrybuty) tego

prostokąta z wartością zmiennej. Często wybieranym do tego celu atrybutem jest wysokość prostokąta. Zadaniem elementu tłumaczącego jest takie przetworzenie wartości zmiennej, by wartości przyjmowane przez tę zmienną w procesie wykonania algorytmu określały we właściwy sposób wysokość prostokąta, a zatem by różne wartości zmiennej powodowały wyświetlanie zauważalnie różnych wysokości, i by z drugiej strony w żadnym normalnie spotykanym stanie algorytmu prostokąt nie stawał się tak wysoki, by nie mieścić się na ekranie.

TRANSLATORY

Translator jest obiektem realizującym funkcję przetwarzającą wartość pobraną z programu poddawanego wizualizacji na taką wartość, którą podczas projekcji przyjmować będzie odpowiedni atrybut grela. Dostępne są różne klasy translatorów, realizujące różne funkcje. Podobnie, jak grele, mają one także własną hierarchię klas.

Translatory są często parametryzowane za pomocą czwórki parametrów (x_0, x_1, y_0, y_1), przy czym dla funkcji f realizowanej przez dany translator spełnione jest: $f(x_0) = y_0$ oraz $f(x_1) = y_1$. Translatory można konfigurować kaskadowo, otrzymując złożenie odpowiednich funkcji.

System *Daphnis* oferuje bogaty zbiór klas translatorów. Do najczęściej używanych należą:

- Translator linearny (ang. *linear translator*). Dla czwórki parametrów (x_0, x_1, y_0, y_1) realizuje funkcję:

$$F(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

- Translator sinusoidalny (ang. *sine translator*). Dla czwórki parametrów (x_0, x_1, y_0, y_1) realizuje funkcję:

$$F(x) = y_0 + (y_1 - y_0)\sin 2\pi \frac{x - x_0}{x_1 - x_0}$$

- Translator kwadratowy (ang. *square translator*). Jest to bezparametrowy translator realizujący funkcję:

$$F(x) = x^2$$

- Translator progowy (ang. *threshold translator*). Dla trójki parametrów (x_t, y_0, y_1) realizuje funkcję:

$$F(x) = \begin{cases} y_0 \ dla \ x < x_t \\ y_1 \ dla \ x \geq x_t \end{cases}$$

- Translatory operatorowe (ang. *operator translators*). Realizują podstawowe operacje arytmetyczne i logiczne, a także relacje.
- Translator odwrotny (ang. *reverse translator*). Współpracuje z innymi translatorami, realizując funkcję odwrotną do funkcji przez nie realizowanej.

Należy zaznaczyć, że podobnie, jak to miało miejsce w przypadku greli, otwarta architektura systemu *Daphnis* pozwala dynamicznie wzbogacać go o nowe klasy translatorów.

ATRYBUTY STAŁE

Często atrybutom greli przypisuje się wartości stałe: odpowiada to tym aspektom elementu graficznego, które nie będą się zmieniać w trakcie projekcji, i nie będą zależne od danych. Atrybuty takie określamy mianem **atrybutów stałych** (ang. *constant attributes*). Jakkolwiek w rzeczywistych projekcjach większa część atrybutów pozostaje niezmienna, o istocie działania wizualizacji decydują oczywiście te atrybuty, które powiązano z danymi.

ATRYBUTY WARTOŚCI

Atrybuty nie będące atrybutami stałymi często są uzależnione od przetworzonej **wartości** zmiennej. Nazywamy je wówczas **atrybutami wartości** (ang. *value attributes*). Rzeczywiste wartości atrybutów są obliczane na podstawie wartości poddawanej wizualizacji zmiennej, przy zastosowaniu wskazanego przez użytkownika translatora.

Gdy wartość poddawanej wizualizacji zmiennej ulega w trakcie procesu wykonania algorytmu zmianie, wówczas obliczana jest nowa wartość atrybutu, a obraz grela ulega odpowiedniej modyfikacji: zgodnie z formułą systemu, modyfikacja ta ma charakter płynnej animacji, przedstawiającej łagodne przejście od poprzedniej do następnej wartości atrybutu.

ATRYBUTY PRZESTRZENNE

Niektóre atrybuty greli mogą też być powiązane z **adresem** zmiennej. Ponieważ najczęściej dotyczy to atrybutów określających położenie obrazu grela, atrybuty powiązane z adresem zmiennej nazywamy **atrybutami przestrzennymi** (ang. *spatial attributes*). Do atrybutów tego rodzaju nie stosuje się translacji: adresy nie są traktowane jak wartości liczbowe. Zamiast tego, atrybutom przestrzennym przyporządkowuje się wartości stałe.

Atrybuty przestrzenne są niezmiernie istotne podczas wizualizacji przepływu danych. Przedmiotem wizualizacji jest wartość, która przepływa od pewnego miejsca źródłowego do pewnego miejsca docelowego, innymi mówiąc słowy, zmienia swój adres. Adres zmiennej

nie może być wprost przeliczony na wartość atrybutu. Zamiast tego wyszukiwany jest element graficzny, który dotąd reprezentował wartość znajdującą się w miejscu docelowym wizualizacji. Wartości wszystkich atrybutów przestrzennych grela reprezentującego wartość wypływającą z miejsca źródłowego są wymieniane na wartości odpowiednich atrybutów grela reprezentującego wartość znajdującą się w miejscu docelowym. Ponieważ najczęściej atrybutami przestrzennymi są atrybuty określające położenie grela na ekranie, zmiana taka prowadzi zwykle do przesunięcia się grela od dotychczasowej pozycji do pozycji związanej ze zmienną docelową; dodajmy – łagodnego przesunięcia, obrazowanego za pomocą płynnej animacji. Oczywiście, wartość znajdująca się w miejscu docelowym jest zacierana; reprezentujący ją grel znika (o ile nie przepływa jednocześnie uprzednio w inne miejsce, jak na przykład podczas zamiany wartości dwóch zmiennych). Dokładniej mechanizm wizualizacji przepływu danych, z wykorzystaniem atrybutów adresowych opisaliśmy w rozdziale 4.4.3.

ATRYBUTY INDEKSOWE

Atrybuty indeksowe są odmianą atrybutów przestrzennych; w identyczny sposób są traktowane podczas wizualizacji przepływów danych. Atrybuty te są powiązane z **indeksem** zmiennej, którą reprezentują (o ile zmienna ta jest elementem tablicy). Od atrybutów przestrzennych odróżnia je fakt, że nadaje im się wartości obliczone na podstawie wartości indeksu zmiennej w tablicy, przy czym do przeliczenia wartości indeksu na wartość wybranego atrybutu grela stosuje się podobne sposoby translacji, jak to miało miejsce w przypadku atrybutów wartości.

6.4.5 SKŁADNIA SKRYPTÓW KONFIGURACYJNYCH

Poniżej zamieszczamy definicję składni skryptu konfiguracyjnego, zapisaną w zmodyfikowanej notacji BNF. W dalszej części tego punktu omówimy tę definicję.

```
skrypt          ::= {aktor}.
aktor           ::= ["noncopy" | "nc"] etykieta "is" grel.
etykieta        ::= identyfikator.
grel            ::= nazwa-klasy [parametry] "{" {atrybut} "}".
nazwa-klasy     ::= identyfikator.
parametry       ::= "(" parametr {"," parametr } ")".
parametr        ::= liczba-całkowita | liczba-zmiennoprzecinkowa
                | łańcuch.
atrybut         ::= {specyfikator} nazwa "=" wyrażenie.
```

```
specyfikator      ::= "value" | "spatial" | "index"
                  | "animated" | "discrete"
                  | "clockwise" | "c" | "counterclockwise" | "cc"
                  | "straight" | "s".
nazwa             ::= identyfikator.
wyrażenie         ::= składnik { op składnik }
                  | "(" wyrażenie ")".
składnik          ::= liczba-całkowita | liczba-zmiennoprzecinkowa
                  | "value" | "x" | "index" | "#" etykieta | translator.
translator        ::= nazwa-translatora parametry.
nazwa-translatora ::= identyfikator.
op                ::= "!"
                  | "*" | "/" | "%"
                  | "+" | "-"
                  | "<" | "<=" | ">" | ">="
                  | "==" | "!="
                  | "&&"
                  | "||"
                  | "?" | ":"
                  | "of".
```

Z punktu widzenia wizualizatora, tworzenie skryptu konfiguracyjnego sprowadza się do zdefiniowania zbioru aktorów. Aktor to para złożona ze zmiennej poddawanego wizualizacji programu oraz grela, będącego jej reprezentacją. Podana w skrypcie konfiguracyjnym etykieta – identyfikator zmiennej – powinna być identyczna z tą, którą zastosowano do rejestracji tej zmiennej na etapie modyfikacji tekstu źródłowego programu. Należy w tym miejscu zauważyć, że w toku projekcji, na skutek przedstawiania operacji przepływu danych, określony grel (aktor) może reprezentować różne zmienne w różnych fazach projekcji.

Definicja aktora może być poprzedzona specyfikatorem **noncopy** (lub w skrócie **nc**). Jego zastosowanie powoduje, że wykrycie operacji przepływu danych, dla której aktor reprezentuje zmienną źródłową, spowoduje likwidację reprezentacji graficznej zmiennej źródłowej dla tego przepływu. Z punktu widzenia użytkownika, przepływająca wartość zostanie przeniesiona na nowe miejsce, a nie skopiowana.

Definicja grela obejmuje nazwę klasy grela, listę parametrów i część najważniejszą – listę atrybutów. Klasy greli zostały omówione w punkcie 6.4.3. Tylko nieliczne z nich wymagają

podania parametrów; dla przykładu grele klasy *bitmap* wymagają podania nazwy pliku zawierającego graficzny wzorzec obiektu.

Definicja atrybutu zawiera opcjonalny specyfikator, nazwę atrybutu i wyrażenie, na podstawie którego obliczana będzie wartość atrybutu. Nazwy dostępnych atrybutów są różne w zależności od klasy grela.

Spośród dopuszczalnych specyfikatorów, trzy precyzują typ atrybutu (por. p. 4.4.3). Są to:

- **value** – oznacza atrybut wartości. Specyfikator ten jest domyślnie przyjmowany dla wszystkich atrybutów z wyjątkiem atrybutów określających położenie grela: *x* i *y*.
- **spatial** – oznacza atrybut przestrzenny. Specyfikator ten jest domyślnie przyjmowany dla atrybutów określających położenie grela: *x* i *y*.
- **index** – oznacza atrybut indeksowy.

Pozostałe specyfikatory określają rodzaj animacji, czyli sposób generowania kolejnych pośrednich wartości atrybutu. Dopuszczalne są następujące specyfikatory:

- **animated** – powoduje, że zmiana wartości atrybutu będzie odzwierciedlana za pomocą płynnie animowanej deformacji obiektu graficznego. Specyfikator ten jest domyślnie przyjmowany dla wszystkich atrybutów.
- **discrete** – wyłącza płynną animację atrybutu, powoduje, że zmiana wartości zostanie odzwierciedlona za pomocą skokowej, jednorazowej zmiany wartości atrybutu.
- **flash** – powoduje, że wartość atrybutu tylko przez chwilę osiągnie nową, wyliczoną wartość, po czym powróci do wartości pierwotnej – uzyskuje się w ten sposób efekt „rozbłysku", szczególnie użyteczny przy operowaniu kolorami elementów graficznych.
- **clockwise** lub **c** – powoduje animację ruchu elementu po łuku, zgodnie z kierunkiem ruchu wskazówek zegara. Uwzględniany tylko dla atrybutów *x* i *y*, określających położenie grela na ekranie. Jeśli nie podano innego specyfikatora, jest przyjmowany domyślnie.
- **counterclockwise** lub **cc** – powoduje animację ruchu elementu po łuku, przeciwnie do kierunku ruchu wskazówek zegara. Uwzględniany tylko dla atrybutów *x* i *y*, określających położenie grela na ekranie.
- **straight** lub **s** – powoduje animację ruchu elementu wzdłuż linii prostej. Uwzględniany tylko dla atrybutów *x* i *y*, określających położenie grela na ekranie.

W przyszłości przewiduje się rozszerzenie systemu pozwalające na to, by użytkownik sam definiował bardziej skomplikowane rodzaje animacji.

Wyrażenie określające sposób obliczania wartości atrybutu jest wykorzystywane przez system w celu skonfigurowania łańcucha translatorów przetwarzających pobrane z programu dane. Do jego zapisu można stosować operatory arytmetyczne i logiczne oraz operatory relacji, które powodują włączenie w łańcuch przekształceń odpowiednich translatorów operatorowych. Honorowane są poziomy priorytetów identyczne, jak w języku *C*. Można używać nawiasów. Dodatkowy operator *of* ma najniższy priorytet, i opiszemy go nieco dalej. Składnikami wyrażeń mogą być następujące symbole:

- Wartości stałe – liczby całkowitoliczbowe lub zmiennoprzecinkowe.
- Słowo **value** lub **x** oznacza wartość zmiennej programu wizualizowanej przez aktualnie przetwarzanego aktora, to jest występującej w nagłówku definicji tego aktora.
- Słowo **index** oznacza, w przypadku aktorów reprezentujących zmienne stablicowane, wartość indeksu zmiennej. Dla zmiennych nie tablicowych symbol ten przyjmuje wartość 0.
- Etykieta (identyfikator) poprzedzony znakiem „#" oznacza wartość zmiennej skojarzonej z tą etykietą, niekoniecznie związanej z aktualnie definiowanym aktorem.
- Translator – oznacza przekształcenie z wykorzystaniem translatora wskazanej klasy. Podaje się nazwę klasy oraz listę ewentualnych parametrów. Klasy translatorów zostały przedstawione w punkcie 6.4.4. Domyślnie, argumentem obliczeń realizowanych przez translator jest zmienna związana z aktualnie definiowanym aktorem. Operator *of* zmienia to ustawienie: wyrażenie znajdujące się po prawej stronie operatora staje się argumentem translatora występującego po lewej stronie operatora. Dzięki temu można nie tylko przetwarzać inne, niż domyślna zmienne, ale także układać translatory w kaskady, uzyskując złożenie odpowiednich funkcji.

Najczęściej wyrażenia definiujące wartość atrybutów składają się z pojedynczej wartości stałej lub translatora. Operatory i nawiasy, a także symbole *value*, *x*, *index* i # są przydatne w bardziej zaawansowanych wizualizacjach.

PRZYKŁAD

Aby zilustrować powyższy opis, przytaczamy prosty skrypt, definiujący jednego aktora. Aktor ten jest przeznaczony do prostej wizualizacji elementów tablicy. Oto treść skryptu:

```
Arr is roundRectangle
{
   index x = linear(0, 1, 10, 30);
   y = 200;
   width = 10;
```

```
    height = linear(0, 100, 50, 200);
    redBrush = 128;
};
```

Tak zdefiniowany aktor przeznaczony jest do reprezentacji zmiennej o symbolicznej nazwie *Arr*; nie musi ona być jednobrzmiąca z identyfikatorem faktycznie użytym w programie, chociaż jest to zalecane. Dokładniej analizując powyższy zapis można zauważyć, że:

- Zmienne będą reprezentowane przez prostokąty o zaokrąglonych rogach (*roundRectangle*).
- Elementy tablicy będą rozłożone w poziomie tak, że element o indeksie *0* znajdzie się w pozycji o współrzędnej $x = 10$, a element o indeksie *1* – w pozycji o współrzędnej $x = 30$. Ponieważ zastosowano translator liniowy, kolejne elementy będą wyświetlane w odległości 20 punktów jeden od drugiego.
- Wszystkie elementy będą wyświetlane na tej samej wysokości $y = 200$.
- Wszystkie elementy będą miały szerokość 10 punktów.
- Wysokość elementów zależy od wartości zmiennej zgodnie z zależnością $h = 50 + \frac{150}{100}x$, gdzie x jest wartością zmiennej.
- Prostokąty będą wypełnione na czerwono (wszystkie pozostałe atrybuty związane z kolorami są domyślnie ustawione na 0).

W połączeniu z przygotowanym do wizualizacji programem sortującym tablicę *Arr*, na przykład takim, jaki zamieściliśmy na stronie 79, nasz skrypt definiuje prostą wizualizację algorytmu sortowania. Przytoczmy jeszcze raz ten program:

```
const int N = 35;
int arr[N];                          Register("arr", arr[0], N);
int i, j;                            Register("i", &i); Register("j", &j);

... // inicjalizacja tablicy

for (i = 0; i < N - 1; i++)          { Play(i);
   for (j = N - 1; j > i; j--)       { Play(j);
        if (arr[j] < arr[j-1])
   {
            int tmp = arr[j]; Register("tmp", &tmp);
                              Play(&tmp, &arr[j]);
            arr[j] = arr[j-1];Play(&arr[j], &arr[j-1]);
            arr[j-1] = tmp;   Play(&arr[j-1], &tmp);
   }                          }}
                              UnRegister(arr[0]);
```

Zauważmy, że zamiana elementów tablicy jest graficznie przedstawiana jako zamiana elementów graficznych miejscami. Dzięki temu już ta wstępna wersja wizualizacji dość do-

brze ilustruje problem sortowania. Zauważmy też, że nasz aktor jest na tyle uniwersalny, że nadaje się do przedstawienia dowolnego algorytmu manipulującego elementami tablicy, a w szczególności – dowolnego algorytmu sortowania.

Spróbujmy uzupełnić skrypt wprowadzając do projekcji więcej informacji charakterystycznych dla algorytmu sortowania bąbelkowego. Specyfikacja aktorów wizualizujących zmienne indeksowe *i* oraz *j* jest banalna i pozostawimy ją bez komentarza:

```
i is rectangle        // zielony prostokąt
{
   x = linear(0, 1, 10, 30);
   y = 450;
   width = 10; height = 40;
   greenBrush = 128;
}
j is rectangle        // czarny prostokąt
{
   x = linear(0, 1, 10, 30);
   y = 450;
   width = 10; height = 40;
};
```

Aksjomatem algorytmu sortowania bąbelkowego jest to, że elementy o indeksach mniejszych niż *i–1* są posortowane i ustawione w swoich końcowych pozycjach. Wyróżnijmy je zatem kolorem zielonym. W tym celu należy dodać nowe wyrażenie dla atrybutu określającego zielony składnik koloru, oraz zmienić wyrażenie dla atrybutu koloru czerwonego. Korzystamy z faktu, że wynikiem operatorów relacji są wartości logiczne *0* lub *1*. Oto dodane lub zmodyfikowane linijki:

```
   greenBrush = 128 * (index < #i);
   redBrush   = 128 * (index >= #i);
```

Ostatnią poprawką niech będzie wprowadzenie krótkiego rozbłysku, wskazującego elementy tablicy poddawane w danej chwili operacji porównania. Dotyczy to elementów o indeksach *j – 1* oraz *j*. Oto ostateczna wersja naszego aktora, w której skorzystaliśmy ze specyfikatora *flash*:

```
Arr is roundRectangle
{
   index x = linear(0, 1, 10, 30);
   y = 200;
   width = 10;
   greenBrush = 128 + (index < #i);
   redBrush   = 128 + (index >= #i);
   flash redBrush = 128 * (index == #j || index == #j - 1);
};
```

Znacznie więcej przykładowych skryptów konfiguracji dla animacji różnych algorytmów można znaleźć w rozdziale 8.

6.5 ŚRODKI WSPOMAGAJĄCE TWORZENIE WIZUALIZACJI

Należy w tym miejscu wyraźnie zaznaczyć, iż opisane dotąd elementy systemu *Daphnis* są w pełni wystarczające do stworzenia dowolnej wizualizacji. Środki wspomagające, które opisujemy niżej, są ułatwieniem, ale ich istnienie nie warunkuje wykonalności.

O ile wszystkie środki opisane w poprzednich podrozdziałach już funkcjonują w obrębie systemu, o tyle środki wspomagające tworzenie wizualizacji w znacznym stopniu znajdują się jeszcze na etapie implementacji lub są dopiero zaplanowane. Tu zaznacza się prototypowy charakter *Daphnis*. Opisywane środki nie mają jednak bezpośredniego związku z tematem niniejszej pracy.

Zaplanowano stworzenie dwóch narzędzi, które wspomagają wizualizatora w toku dwóch najważniejszych czynności, na jakie można zdekomponować proces tworzenia wizualizacji. Narzędzia te to:

- Preprocesor tekstu źródłowego programu poddawanego wizualizacji. Biorąc pod uwagę fakt, że w *Daphnis* bardzo wyraźnie ograniczono różnorodność dodatków, o jakie należy uzupełnić tekst źródłowy, stworzenie tego narzędzia nie przedstawia poważniejszej trudności. Obecnie trwa implementacja, w której korzystamy z popularnych narzędzi typu *lex* i *yacc* [199, 200, 204].
- Edytor graficzny, pozwalający na tworzenie skryptów konfiguracji metodą bezpośredniej manipulacji obiektami graficznymi i interaktywnego dialogu z użytkownikiem. Narzędzie to jest jeszcze w sferze planów. Interesującym pomysłem byłaby próba automatycznego proponowania przez system najbardziej adekwatnego sposobu wizualizacji wskazanych przez użytkownika struktur danych. Byłby to duży krok w kierunku systemu automatycznej wizualizacji, będącego być może skutecznym i silnym narzędziem w ręku projektanta oprogramowania.

Ponieważ wizualizacja programu jest zadaniem wykonywanym w bezpośrednim powiązaniu z pisaniem programu, który ma być poddany wizualizacji, zdecydowaliśmy się nie tworzyć żadnego zintegrowanego środowiska pracy dla wizualizatora, a zamiast tego skorzystać z istniejącego środowiska kompilatora. Większość takich środowisk, jak na przykład *Microsoft Visual C++*, pozwala na poszerzanie ich o dodatkowe moduły.

6.6 PODSUMOWANIE – UŻYTECZNOŚĆ SYSTEMU *DAPHNIS*

Daphnis jest systemem nowym, i nie wyszedł jeszcze poza stadium zaawansowanego prototypu. Jest to powodem, dla którego nie przeprowadzono dotąd badań empirycznych nad jego użytecznością. Ocena walorów użytkowych systemu jest dodatkowo utrudniona przez fakt, iż nie jest jeszcze zakończona implementacja niektórych elementów wspomagających pracę wizualizatora. Mimo tych trudności, zamieszczamy poniżej próbę analizy i oceny użyteczności systemu. Przedstawimy też perspektywy i wnioski na przyszłość. Należy się liczyć z tym, że poniższa analiza może w wielu aspektach uwzględniać subiektywny punkt widzenia autora oprogramowania, i powinna być bezwzględnie zweryfikowana studiami empirycznymi.

Użyteczność oprogramowania (ang. *usability*) jest przez wielu autorów określana w podobny sposób (zob. np. [186, 190, 193]). My zastosujemy ujęcie Theo Mandela [192], który stosuje przytoczone poniżej cztery kryteria.

- **Przydatność** (ang. *usefulness*). Z punktu widzenia użytkowników – obserwatorów projekcji, system *Daphnis* pozwala na prezentację przygotowanych wcześniej projekcji, łącznie z możliwością interakcji z oglądanym obrazem. Kluczowym zagadnieniem jest repertuar dostępnych standardowo, gotowych do projekcji algorytmów. Obecnie jest on jeszcze stosunkowo wąski. W niedalekiej przyszłości planujemy szybkie rozszerzenie tego repertuaru. Do prac tych zostaną włączeni studenci. W toku tych prac uzupełniony zostanie zestaw dostępnych klas greli i translatorów – elementów decydujących w kwestii formy możliwych do uzyskania wizualizacji. W ten sposób poszerzymy przydatność systemu dla wizualizatorów. Podsumowując, przydatność systemu, nawet, jeśli nie spełnia jeszcze oczekiwań pewnych grup potencjalnych użytkowników, może być stosunkowo łatwo poszerzona dzięki otwartej architekturze systemu.
- **Efektywność** (ang. *effectivenes*). Interfejs użytkownika systemu *Daphnis* był projektowany z myślą o efektywnym wykorzystaniu go przez użytkowników – obserwatorów projekcji. Uruchomienie wybranej wizualizacji jest bardzo łatwe (zastosowano technologię tzw. kreatorów – ang. *wizards*), a interakcja podczas samej projekcji sprowadza się do niewielu prostych operacji, które mimo to pozwalają dość dokładnie kontrolować przebieg pokazu. Co do interfejsu wizualizatora, potencjalna możliwość wprowadzania ułatwień jest ograniczona przez samą naturę procesu projektowania wizualizacji. Bardzo proste projekcje można wprawdzie w znacznym stopniu zautomatyzować (zwiększając znacznie efektywność systemu dla użytkowników – inżynierów i projektantów oprogramowania), jednak

bardziej zaawansowane wizualizacje zawsze będą wymagać aktywnego projektowania. Należy w tym miejscu zaznaczyć, że ułatwienie pracy wizualizatora stanowi najważniejszy – naszym zdaniem – atut systemu *Daphnis*, i było głównym celem niniejszej pracy. Przy stosunkowo niewielkim nakładzie pracy użytkownik w sposób automatyczny otrzymuje projekcje charakteryzujące się takim stopniem abstrakcji (i zaawansowania), iż dla ich stworzenia przy pomocy innych systemów wizualizacji trzeba by włożyć znacznie więcej wysiłku. Dodatkowo, uczyniliśmy wszystko, co było w naszej mocy, by uczynić składnię skryptu konfiguracyjnego tak prostą, jak to możliwe. Zdajemy sobie jednak sprawę, że efektywność pracy wizualizatora wzrośnie znacznie dopiero wtedy, gdy ukończone zostaną moduły wspomagające jego pracę – preprocesor tekstu źródłowego i edytor graficzny.

- **Przyswajalność** (ang. *learnability*). Interfejs użytkownika – obserwatora projekcji jest na tyle łatwy w obsłudze, że w zasadzie nie wymaga wstępnego treningu (zgodnie z zasadą dobrze zaprojektowanego oprogramowania – podręcznik użytkownika można przed przeczytaniem wrzucić do kosza). Forma graficzna zestawu najważniejszych narzędzi pozwalających użytkownikowi na interakcję z systemem odwołuje się do powszechnie znanej metafory sprzętu audio-video.

  Interfejs wizualizatora, a przede wszystkim – zasady tworzenia i składnia skryptów konfiguracyjnych – wymaga przyswojenia sobie podstawowych wiadomości na temat organizacji projekcji. Spodziewamy się, że składnia skryptów jest dość łatwo przyswajalna dla programistów znających język *C*, gdyż w znacznym stopniu nawiązuje do składni tego języka. Podobnie jak w przypadku efektywności, tak i pod względem przyswajalności stworzenie narzędzi wspomagających pracę wizualizatora znacznie uatrakcyjni system.
- **Stosunek użytkownika** do oprogramowania (ang. *attitude*). Wydaje się uzasadniony sąd, iż przyjęte w systemie *Daphnis* rozwiązania w zakresie interfejsu użytkownika nie spowodują negatywnych reakcji – szczególnie w zakresie interfejsu obserwatora. Mamy też nadzieję, że tworzone wizualizacje mają na tyle atrakcyjną formę, by skupić uwagę obserwatora. Mamy w tym zakresie pewne doświadczenia z poprzednikami systemu *Daphnis* – systemami *WinSANAL* i *SANAL* (por. rozdział 3.3, str. 54 – 57). Były one prezentowane na kilku imprezach targowych (między innymi trzykrotnie na międzynarodowych targach CEBIT w Hanowerze), i wzbudzały zainteresowanie osób odwiedzających stoisko Politechniki Śląskiej. W kilku przypadkach dotyczyło to nauczycieli informatyki.

Co do stosunku użytkowników do interfejsu wizualizatora, trudno na razie cokolwiek zawyrokować, gdyż wszystkie przygotowane dotąd wizualizacje zostały zrealizowane wyłącznie przez autora niniejszej pracy.

Na zakończenie jeszcze raz bardzo wyraźnie podkreślimy, że przedstawiona powyżej ocena użyteczności systemu *Daphnis* ma charakter wstępny i wymaga empirycznej weryfikacji. W szczególnym stopniu dotyczy to oceny przyswajalności systemu, a także stosunku użytkowników. Jak już wspomnieliśmy, w najbliższym czasie przewidujemy włączenie grup studenckich do tworzenia nowych animacji algorytmów, w tym i takich, które wymagają sięgnięcia po możliwości wynikające z otwartej architektury systemu. Wydaje się, że będzie to idealna sposobność do przeprowadzenia koniecznych badań empirycznych.

7
ORGANIZACJA SYSTEMU DAPHNIS

Przejdziemy obecnie do opisu wewnętrznej struktury systemu *Daphnis*. Najogólniej można ją scharakteryzować odwołując się do modelu *POST*. W dalszej części rozdziału w sposób bardziej szczegółowy opiszemy poszczególne elementy systemu, a także sposób połączenia systemu z poddawanym wizualizacji programem oraz możliwości systemu *Daphnis* jako systemu otwartego.

Zawartość niniejszego rozdziału wykracza poza zakres wiadomości przydatny osobom chcącym tworzyć własne wizualizacje. Konieczny w tym zakresie opis przebiegu projekcji i jej elementów strukturalnych czytelnik znajdzie w rozdziale 6.4. Użytkownikom poszerzającym repertuar środków oferowanych przez system, korzystającym z jego otwartej architektury, mogą natomiast być przydatne informacje zawarte w rozdziale 7.7.

7.1 ORGANIZACJA SYSTEMU *DAPHNIS* A MODEL *POST*

Początkowo, ogólny projekt maszyny projekcyjnej systemu *Daphnis*, a właściwie jego poprzedników, systemów SANAL i WinSANAL określaliśmy mianem modelu warstwowego. Zakładał on podział strukturalny maszyny na trzy moduły, zwane warstwami:

- warstwę obserwacji, odpowiedzialną za ekstrakcję danych z programu poddawanego wizualizacji,
- warstwę translacji, odpowiedzialną za przetworzenie surowych danych, pozyskanych w obrębie warstwy obserwacji,
- warstwę wyświetlania, odpowiedzialną za przedstawienie przetworzonych danych w postaci graficznej.

W 1997 roku, pod wpływem prac F. Van de Veire'a i M. Mostefaia sięgnęliśmy po model *POST* (Prezentacja, Obiekt Symulacji, Translator) [31, 32, 116, 120]. Nie wymagało to

zresztą dokonania znaczniejszych zmian w samym systemie, gdyż oba podejścia są w znacznym stopniu zbieżne.

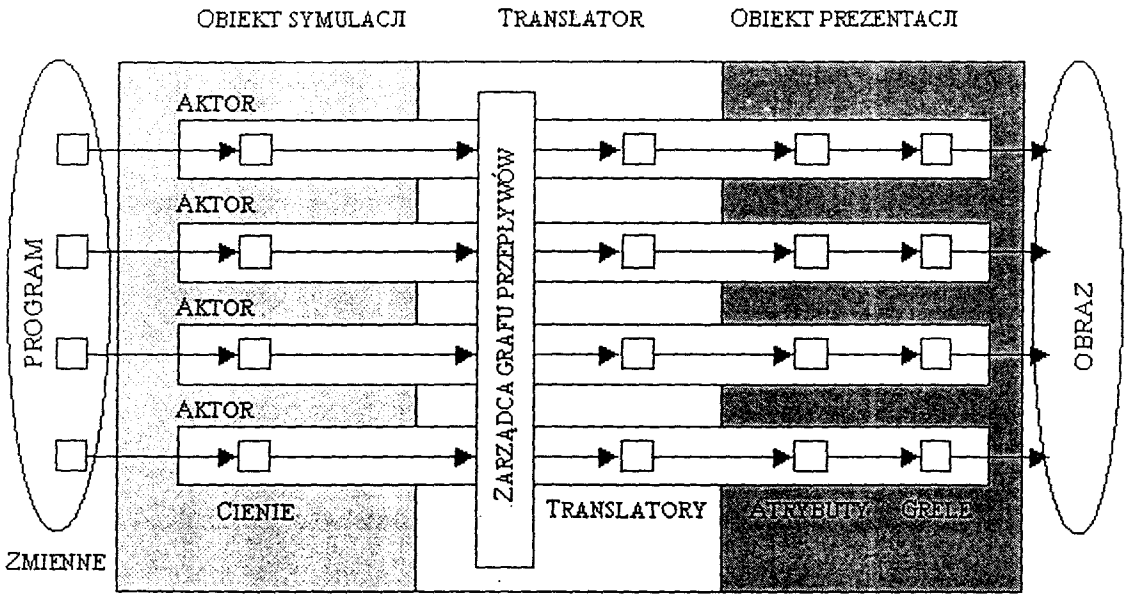
Model *POST* został pierwotnie stworzony jako narzędzie do tworzenia interaktywnych platform symulacji i wizualizacji procesów przemysłowych i był zaimplementowany w środowisku Smalltalk [31, 32]. Miał zapewnić możliwość tworzenia wizualnych interfejsów człowiek – maszyna w środowisku przyjaznym dla użytkownika, bez konieczności znajomości zaawansowanych języków graficznych. Model *POST* był też już adaptowany dla potrzeb wizualizacji algorytmów [116, 120].

Maszyna projekcyjna systemu *Daphnis* stanowi alternatywną implementację dla modelu *POST*, stworzoną w języku C++. Scharakteryzujemy trzy elementarne składowe architektury *POST* w systemie *Daphnis*:

- Obiekt symulacji, reprezentujący w przypadku systemu animacji poddawany wizualizacji program. W systemie *Daphnis* obiekt symulacji ma dwa aspekty (co jest zresztą zgodne z ogólną koncepcją modelu *POST*). Są to:
  - aspekt statyczny – obejmuje on informacje dotyczące występujących w danej chwili wartości poszczególnych zmiennych obserwowanych w programie,
  - aspekt dynamiczny – obejmuje on informacje dotyczące wykrytych w systemie operacji przepływu danych.
- Obiekt prezentacji odpowiada za wizualną stronę całej projekcji. Również i ten obiekt obejmuje dwa aspekty:
  - aspekt statyczny, opisujący docelową formę poszczególnych reprezentacji graficznych używanych w projekcjach, która to forma odnosi się wprost do stanów występujących w obiekcie symulacji,
  - aspekt dynamiczny, opisujący animowane, płynne przejścia pomiędzy kolejnymi stanami graficznych reprezentacji danych, które nie mają bezpośredniego odniesienia ani do stanów występujących w obiekcie symulacji, ani też do dyskretnych w swym charakterze zmian tych stanów.
- Translator, który przetwarza dane otrzymane z obiektu symulacji tak, by mogły one we właściwy sposób sterować obrazem graficznym wyświetlanym przez moduł prezentacji. W systemie *Daphnis* wyróżniamy dwa kierunki translacji:
  - translację zwykłą, generującą dane dla modułu prezentacji,

- translację odwrotną, pozwalającą na zwrotną interpretację wielkości spotykanych w module prezentacji i odnoszenie ich do obiektu symulacji; translacja odwrotna umożliwia graficzne sterowanie obiektem symulacji.

W systemie *Daphnis* model *POST* można zidentyfikować na poziomie globalnym oraz lokalnym (rys. 7.1). W **modelu globalnym**, do poszczególnych części modelu *POST* zaliczamy całe moduły systemu. Jest to podział poziomy architektury systemu, nawiązujący do dawniejszej koncepcji modelu warstwowego. W istocie trzy moduły modelu *POST* odpowiadają trzem warstwom modelu warstwowego.



Rys. 7.1. Globalny model oraz lokalne modele *POST* w systemie *Daphnis*

Technicznie rzecz biorąc, na podstawowe części wyróżniane w modelu *POST* składają się następujące klasy występujące w architekturze systemu:

- Moduł obiektu symulacji:
  - za aspekt statyczny jest odpowiedzialna grupa obiektów klasy cieni (*CShadow*),
  - za aspekt dynamiczny odpowiada obiekt zarządcy grafu przepływu (*CFlowChart*).
- Moduł obiektu prezentacji:
  - aspekt statyczny jest kontrolowany przez obiekty klasy greli (*CGrel*),
  - aspekt dynamiczny realizują animatory (*CAnimator*).
- Moduł translatora obejmuje klasy translatorów (*CTranslator*) i atrybutów (*CAttribute*).

Za całokształt organizacji projekcji odpowiadają obiekty klasy sceny (*CScene*). Można powiedzieć, że implementują one model *POST* na poziomie globalnym

W **modelu lokalnym *POST*** analizujemy ścieżkę transformacji danych prowadzącą od określonej jednostkowej informacji, wyekstrahowanej z poddawanego wizualizacji programu, aż po nadanie tej informacji przeznaczonej dla niej szaty graficznej. Prowadzi ona od obiektu cienia (*CShadow*), który dostarcza systemowi informacji dotyczących statycznej zawartości zmiennej poddawanej wizualizacji. Stanowi on obiekt symulacji w sensie modelu *POST*. Ponadto model lokalny obejmuje obiekty przetwarzające dane, stanowiące moduł translatora. Obiekty te to atrybuty (*CAttribute*) i stowarzyszone z nimi obiekty – translatory (*CTranslator*). Wreszcie, lokalny model *POST* obejmuje wybrany do reprezentacji danych obiekt graficzny, tzw. grel – stanowiący obiekt prezentacji. Takiemu kompletnemu, lokalnemu schematowi POST odpowiada obiekt klasy aktor (*CActor*), który grupuje wszystkie wymienione powyżej elementy.

7.2 ARCHITEKTURA SYSTEMU – WSTĘP

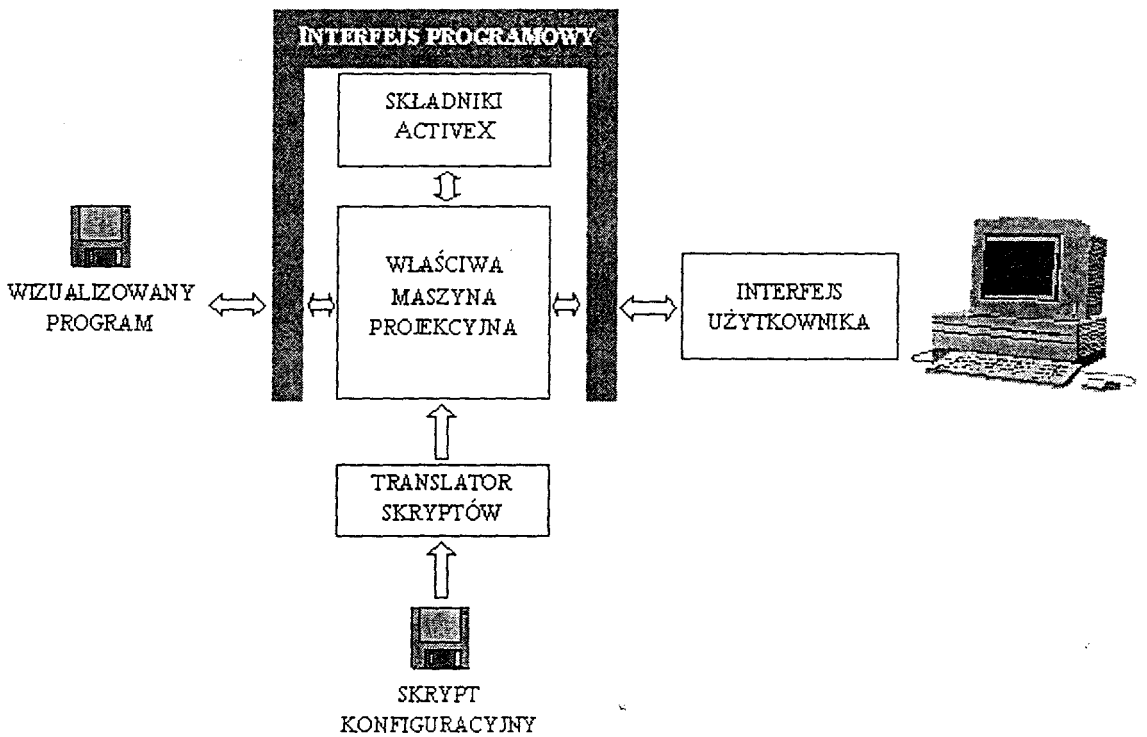
W rozdziale 5.3 przedstawione zostały elementy konstrukcyjne systemu *Daphnis*. Mówiąc o architekturze systemu zajmować się będziemy tylko tą jego częścią, która stanowi **maszynę projekcyjną**, to jest ogół środków pozwalających na przeprowadzenie i wyświetlenie projekcji. Pominiemy zatem moduł programowy *DAPHNIS.EXE*, ułatwiający użytkownikowi wybór i uruchomienie interesującej go projekcji, narzędzia wspomagające pracę wizualizatora, a także zestaw przykładowych programów przygotowanych do projekcji.

W obrębie maszyny projekcyjnej można wyróżnić kilka modułów (rys. 7.2). Wśród nich wyróżniamy **właściwą maszynę projekcyjną** – najważniejszą część całego systemu. Odpowiada ona za całokształt przebiegu projekcji: pobranie danych z poddawanego wizualizacji programu, ich przekształcenie i wyświetlenie. Pozostałe moduły stanowią szeroko rozumiany interfejs pozwalający na funkcjonowanie właściwej maszyny projekcyjnej w jej otoczeniu. Poniżej krótko je scharakteryzujemy:

- **Interfejs programowy** (ang. *application programmers interface, API*) to biblioteka funkcji usługowych dostępnych dla programów poddawanych wizualizacji oraz dla modułu interfejsu użytkownika.
- **Translator skryptów** to moduł stanowiący interfejs pomiędzy maszyną projekcyjną a skryptem konfiguracyjnym. Wczytuje on zawartość skryptu, dokonuje jego leksykalnej,

składniowej i semantycznej analizy, i na jej podstawie steruje wytworzeniem w obrębie maszyny projekcyjnej odpowiednich obiektów – prototypów, które podczas projekcji służą jako wzorce do tworzenia właściwych obiektów biorących udział w pokazie.

- **Interfejs użytkownika** to moduł odpowiedzialny za wyświetlenie okna projekcyjnego i interakcję z użytkownikiem. Za pośrednictwem interfejsu programowego (patrz: wyżej) moduł pozwala użytkownikowi sterować przebiegiem projekcji. Omawiany moduł jest też odpowiedzialny za skalowanie i przewijanie (przesuwanie) wyświetlanego obrazu. Sam jednak proces wyświetlania (rysowania) leży poza zakresem kompetencji modułu, który ogranicza się jedynie do udostępnienia właściwej maszynie projekcyjnej kontekstu urządzenia (ang. *device context*), reprezentującego zawartość okna projekcyjnego.
- **Składniki *ActiveX*** [203, 205 – 208] stanowią integralną część właściwej maszyny projekcyjnej. Ich lokalizacja (w osobnych plikach typu *dll*) i sposób połączenia z resztą systemu zapewnia łatwą rozbudowę, czyniąc z *Daphnis* system otwarty. Uzupełnienie systemu o nowe składniki nie wymaga dostępu do tekstów źródłowych *Daphnis*, ani nawet ich znajomości.



Rys. 7.2. Podział maszyny projekcyjnej systemu *Daphnis* na moduły

Właściwa maszyna projekcyjna, interfejs programowy i translator skryptów umieszczono we wspólnym pliku o nazwie *Daphnis.dll*. Udało się w znacznym stopniu zachować ich nie-

zależność od środowiska systemu operacyjnego (*Microsoft Windows*), dzięki czemu przewidujemy wysoki stopień przenośności tych modułów. Fragmenty odwołujące się do architektury konkretnego systemu operacyjnego zajmują nie więcej, niż 5% tekstu źródłowego.

Moduł interfejsu użytkownika rezyduje w osobnym pliku, *DaphnisWin.dll*. Z technicznego punktu widzenia jest on trywialny: wszystkie opcje zaimplementowane w module (z wyjątkiem skalowania i przesuwania obrazu) są dostępne wprost w module interfejsu programowego maszyny projekcyjnej. W ramach tego modułu nie staraliśmy się zachować niezależności od platformy, w przypadku jednak przenoszenia systemu do innego środowiska nawet stworzenie modułu od podstaw byłoby zadaniem stosunkowo łatwym.

Składniki *ActiveX* rezydują – oczywiście – w osobnych plikach. Jednym z potencjalnych problemów, jakie powstać mogą przy ewentualnym przenoszeniu systemu *Daphnis*, jest zgodność nowego środowiska z technologią *ActiveX*. Należy jednak zauważyć, że gwałtowna ekspansja tej technologii obejmuje także jej implementację na innych niż *Windows* platformach, wliczając w to środowiska systemu *UNIX*.

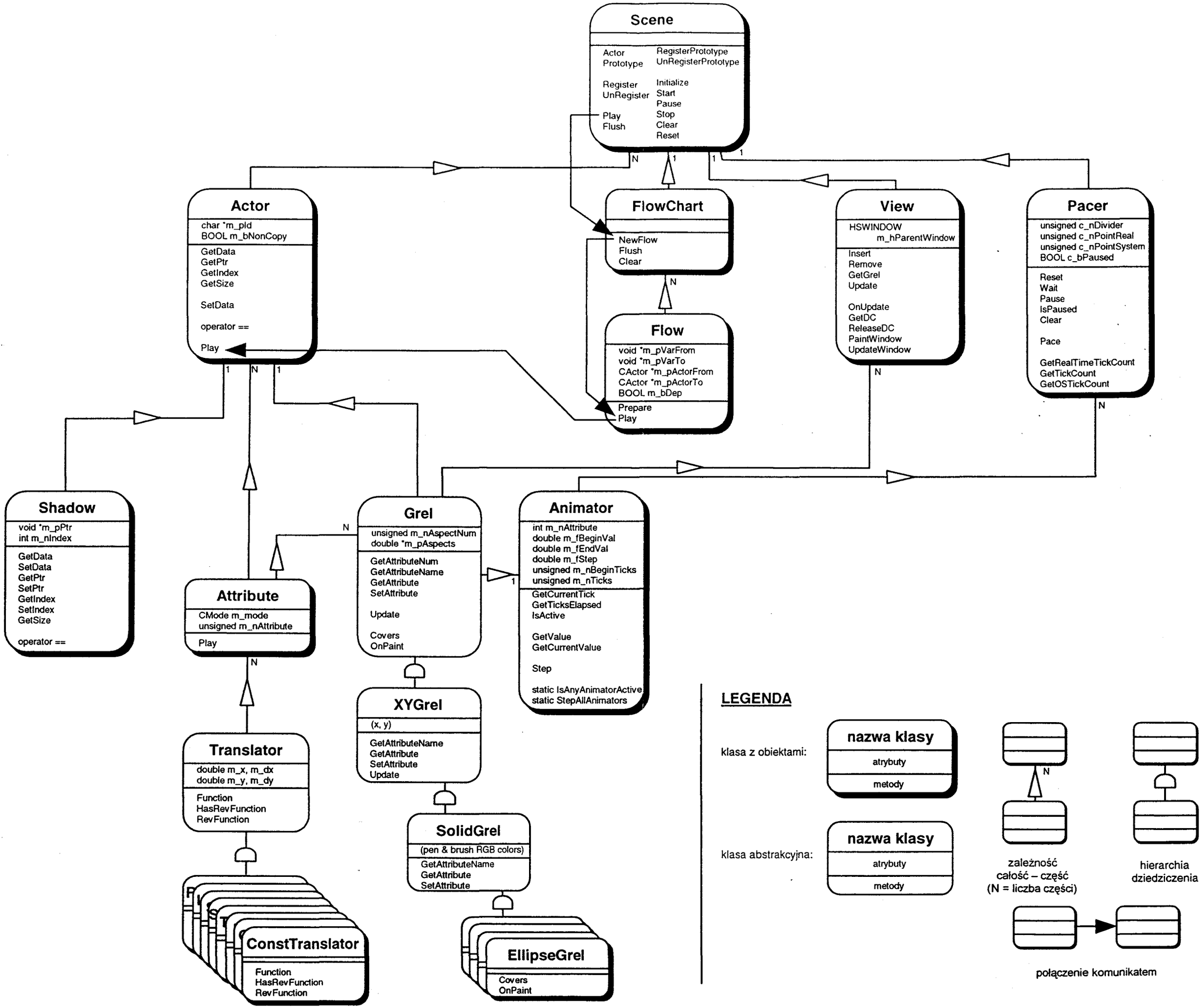
W kolejnych podrozdziałach opiszemy pięć wymienionych wyżej modułów maszyny projekcyjnej systemu *Daphnis*.

7.3 WŁAŚCIWA MASZYNA PROJEKCYJNA

Schemat organizacji właściwej maszyny projekcyjnej systemu *Daphnis* przedstawia plan sza 7.1. Schemat ten powstał w wyniku zastosowania metod analizy i projektowania obiektowego. Uwzględnia on też wszystkie zmiany wprowadzone na etapie implementacji. Zastosowana notacja graficzna pochodzi od Coada i Yourdona [188, 189].

W ramach właściwej maszyny projekcyjnej zidentyfikowano jedenaście klas. W tej liczbie dwie są klasami abstrakcyjnymi i posiadają specjalizacje (klasy potomne). W obydwu przypadkach zestaw klas potomnych jest otwarty – użytkownik może wprowadzać do system własne klasy – specjalizacje. W tab. 7.1 zebrano podstawowe informacje dotyczące głównyc klas maszyny projekcyjnej.

W organizacji właściwej maszyny projekcyjnej systemu *Daphnis* przeważają struktury typu całość – część, wiążące obiekty poszczególnych klas. Na szczycie całej struktury znajduje się globalny obiekt **sceny**, zarządzający całokształtem projekcji. Jak wspomnieliśmy, implementuje on model *POST* w ujęciu globalnym. Do najważniejszych – z punktu widzenia a



Plansza 7.1. Schemat organizacji maszyny projekcyjnej systemu *Daphnis*

chitektury systemu – składowych sceny należą **aktorzy** – obiekty sterujące wizualizacją poszczególnych zmiennych programu i implementujące model *POST* w ujęciu lokalnym.

Scena	*CScene*	Zarządza całokształtem projekcji	obiekt globalny
Widok	*CVirtualView*	Reprezentuje obszar roboczy okna projekcyjnego	obiekt globalny
Generator sekwencji	*CPacer*	Steruje tempem projekcji i odmierza czas systemowy	obiekt globalny
Zarządca grafu przepływów	*CFlowChart*	Ustala sekwencję przedstawiania operacji przepływu. Realizuje algorytm przestrzennej i temporalnej redukcji informacji	obiekt globalny
Przepływ	*CFlow*	Reprezentuje przeznaczoną do wizualizacji jednostkową operację przepływu danych	
Aktor	*CActor*	Odpowiada za wizualizację pojedynczej zmiennej	
Cień	*CShadow*	Pobiera lub modyfikuje wartość zmiennej w poddawanym wizualizacji programie	
Atrybut	*CAttribute*	Reprezentuje atrybut obiektu graficznego, sterowany przez wizualizowane wartości	
Translator	*CTranslator*	Przekształca dane pobrane z programu	klasa abstrakcyjna
Grel	*CGrel*	Obiekt graficzny – graficzna reprezentacja danych	klasa abstrakcyjna
Animator	*CAnimator*	Steruje płynną animacją obiektów graficznych (greli)	

Tab. 7.1. Zestawienie podstawowych klas maszyny projekcyjnej systemu *Daphnis*

Scena i aktorzy – nazwy klas kluczowych dla całego procesu sterowania projekcją – odwołują się do metafory teatralnej. Konwencji tej nie utrzymaliśmy dla pozostałych klas, nadając im nazwy najlepiej oddające ich funkcję w systemie. Próbując jednak poszerzyć teatralną przenośnię na pozostałe składniki, moglibyśmy powiedzieć, że reżyserem, czy też raczej choreografem, jest obiekt zarządcy grafu przepływów, dekoracje to obiekt widoku, zaś grele, atrybuty, translatory i animatory pełnią rolę rekwizytów. Wizualizowany program wraz z obiektami cieni stanowiłby w tej konwencji tekst sztuki.

Poza strukturami typu całość – część, w organizacji maszyny projekcyjnej wyróżnić możemy też hierarchie dziedziczenia, czyli struktury typu uogólnienie – specjalizacja. mamy z nimi

- **Grelem** (*CGrel*) – podstawowy element graficzny (ang. *GRaphical ELement*) – odpowiada za wyświetlenie stosownej, graficznej reprezentacji zmiennej. Wspominaliśmy już o grelach jako o elementach składowych obiektu widoku. Klasy – specjalizacje abstrakcyjnej klasy greli tworzą elementy graficzne o rozmaitych kształtach i właściwościach, a nawet elementy nie mające jawnej postaci graficznej, przeznaczone do udźwiękowienia wizualizacji.
- Kolekcja **atrybutów** (*CAttribute*), z których każdy reprezentuje określoną właściwość (atrybut) grela – na przykład położenie, rozmiar czy kolory. Obiekt atrybutu jest odpowiedzialny za nadanie właściwości grela wartości odpowiednio obliczonej na podstawie wartości danej udostępnionej przez aktora. W procesie obliczania tej wartości biorą udział **translatory** (*CTranslator*) – obiekty przekształcające dane. Za ich pomocą konstruuje się całe ścieżki transformacji danych, odpowiadające wyrażeniom umieszczonym w skrypcie konfiguracyjnym projekcji. Wykorzystuje się tu fakt, że translatory dają się konfigurować w postaci struktur łańcuchowych, także rozgałęzionych.
- **Animator** (*CAnimator*) – obiekt odpowiedzialny za graficzną stronę procesu transformacji greli, czyli nadawania poszczególnym ich atrybutom nowych wartości. Obiekty te sterują płynną animacją greli.

Podsumowując, wizualizacja zmiany stanu określonej zmiennej odbywa się według następującego scenariusza: aktor pobiera za pomocą cienia wartość wizualizowanej zmiennej, i przekazuje ją kolejno wszystkim atrybutom. Każdy z atrybutów, korzystając z łańcucha translatorów oblicza nową wartość określonej właściwości i przekazują ją do grela. Grel tworzy dynamicznie związany z nim obiekt animatora, który steruje płynną animacją transformacji związanej ze zmianą wartości atrybutu.

7.4 INTERFEJS PROGRAMOWY

Technicznie rzecz ujmując, interfejs programowy jest zaimplementowany w postaci zestawu funkcji figurujących na liście eksportowej biblioteki *Daphnis.dll*. Obejmuje on zarówno funkcje przeznaczone do wykorzystania przez poddawane wizualizacji programy, jak i funkcje udostępniane modułowi interfejsu użytkownika *DaphnisWin.dll*. Obydwie grupy funkcji opiszemy osobno.

7.4.1 INTERFEJS DLA MODUŁU *DAPHNISWIN.DLL*

Deklaracje funkcji maszyny projekcyjnej przeznaczonych do wykorzystania przez moduł interfejsu użytkownika (*DaphnisWin.dll*) zebrano w pliku nagłówkowym *GUI.h*. Obejmuje on 21 funkcji, które podzielić można na następujące grupy tematyczne:

- Funkcja inicjalizacji projekcji. Interfejs użytkownika musi udostępnić właściwej maszynie projekcyjnej 32-bitowy uchwyt okna wizualizacji.
- Funkcja odczytu pliku skryptu konfiguracyjnego.
- Tzw. funkcje podtrzymania – ich regularne wywoływanie przez moduł interfejsu użytkownika jest warunkiem prawidłowej pracy systemu. Chodzi tu o wymienione niżej dwie funkcje.
  - Funkcja *PaintAnimation* – wywoływana przez interfejs użytkownika za każdym razem, gdy potrzebne jest odtworzenie (narysowanie) zawartości okna projekcyjnego. W toku projekcji rysowanie okna jest najczęściej prowokowane przez samą maszynę projekcyjną, która wymusza odświeżenie zawartości okna po każdej zmianie konfiguracji obrazu.
  - Funkcja *Pace* podtrzymuje pracę systemu i z założenia powinna być wywoływana przez moduł interfejsu użytkownika w ramach obsługi przerwania zegarowego.
- Funkcje interfejsu użytkownika – realizują wprost wszystkie opcje dostępne z poziomu menu oraz paska narzędziowego modułu interfejsu użytkownika. Wyjątkiem są opcje skalowania i przewijania obrazu.
- Funkcje stanu interfejsu użytkownika – zwracają informacje zależne od stanu systemu, na przykład określające tryby pracy i dostępność poszczególnych opcji w danej chwili. Wykorzystywane przez moduł interfejsu między innymi do tzw. wyszarzania nieaktywnych opcji, sygnalizacji trybu pauzy itd.

7.4.2 INTERFEJS PROGRAMU UŻYTKOWNIKA

Interfejs programu użytkownika zawiera funkcje, których wywołania można lub trzeba umieścić w tekście poddawanego wizualizacji programu. Deklaracje funkcji (dla programów napisanych w językach *C* lub *C++*) zebrano w pliku nagłówkowym *daphnis.h*.

System *Daphnis* jest niezależny od języka programowania, w którym zapisano poddawany wizualizacji program. Musi to jednak być język dający możliwość wywoływania funkcji zebranych w 32-bitowych bibliotekach dynamicznych systemu *Windows* (*dll*). Należy zauważyć, że warunek ten spełniają prawie wszystkie języki zaimplementowane w tym środowisku.

W zależności od użytego języka, wizualizator może użyć różnych sposobów dostępu do interfejsu programowego systemu *Daphnis*.

I tak, w najbardziej komfortowej sytuacji znajdują się użytkownicy piszący w języku *C++*. Nie tylko są oni w stanie wprost skorzystać z pliku nagłówkowego *daphnis.h*, ale otrzymują interfejs w najwygodniejszej postaci. Skorzystano z przestrzeni nazewniczej (ang. *namespace*), wykluczając możliwość wystąpienia konfliktów nazw pomiędzy programem a systemem. Uproszczono też nazewnictwo poszczególnych funkcji korzystając z przeciążania. W nieco gorszej sytuacji są użytkownicy piszący w języku *C*. Wprawdzie i oni mogą wprost skorzystać z pliku nagłówkowego *daphnis.h* (zawiera on dyrektywy kompilacji warunkowej, pozwalające odróżnić kompilację w środowisku *C* i *C++*), ale nie mają możliwości skorzystania ani z przestrzeni nazewniczej, ani z przeciążania funkcji. Komplikuje to nieco i wydłuża udostępniane im nazwy funkcji. W najmniej dogodnej sytuacji są użytkownicy korzystający z innych języków programowania: do ich dyspozycji oddano te same funkcje, które udostępniono programistom w *C*, jednak zmuszeni są oni we własnym zakresie dokonać wszystkich potrzebnych deklaracji importowych funkcji.

Wyczerpujący przegląd funkcji udostępnianych przez interfejs programu użytkownika został przedstawiony w rozdziale 6.4.1, i nie będziemy go tu powielać.

7.5 TRANSLATOR SKRYPTÓW

Translator skryptów, moduł interfejsu między maszyną projekcyjną a skryptem konfiguracyjnym, dokonuje leksykalnej, składniowej i semantycznej analizy skryptu. Steruje on też procesem tworzenia w obrębie właściwej maszyny projekcyjnej prototypów, stanowiących reprezentację skryptu wykorzystywaną podczas projekcji.

Składnię plików skryptowych zdefiniowano w rozdziale 6.4.5.

System *Daphnis* zawiera analizator leksykalny i parser plików skryptowych – obydwa zrealizowane przy pomocy popularnych narzędzi typu *lex* i *yacc* [199, 200, 204]. Bloki akcyjne specyfikacji narzędzia *yacc*, a zatem fragmenty programu parsera wykonywane na skutek wykrycia określonych struktur składniowych, zawierają odwołania do **warsztatu** – obiektu specjalnej klasy *CWorkshop*. Odpowiada on za utworzenie prototypów aktorów. Jednocześnie sprawdza on poprawność odniesień skryptu konfiguracyjnego zależnych od systemu. W szczególności sprawdza:

- poprawność nazw klas i listy parametrów greli,

- poprawność nazw atrybutów greli, w kontekście przynależności tychże greli do klas,
- poprawność nazw klas i listy parametrów klas translatorów.

W trakcie analizy skryptu konfiguracyjnego jest zapewniona elementarna obsługa błędów, przy czym w zakresie błędów składniowych nie jest ona rozbudowana. Lepiej diagnozowane są błędy logiczne, wykrywane przez obiekt warsztatu.

Odczyt i analiza pliku skryptowego są wykonywane każdorazowo przed podjęciem projekcji. Poprawne zakończenie analizy jest warunkiem rozpoczęcia projekcji. W trakcie trwania projekcji, cała informacja pobrana ze skryptu konfiguracyjnego jest wewnętrznie reprezentowana przez kolekcję prototypów aktorów.

Translator skryptów, a konkretnie obiekt warsztatu, jest każdorazowo przy uruchamianiu systemu dynamicznie konfigurowany tak, by mógł uwzględniać obiekty klas zdefiniowanych przez użytkownika. Definicje takich klas umieszczane są w składnikach *ActiveX*. Aby jednak system mógł zostać odpowiednio skonfigurowany, definicje klas muszą być dodatkowo zgłoszone w rejestrze systemowym *Windows* (ang. *system registry*). Szczegóły tworzenia klas obiektów w składnikach *ActiveX* i rejestrowania ich w rejestrze systemowym omawiamy w rozdziale 7.7.

7.6 MODUŁ INTERFEJSU UŻYTKOWNIKA

Moduł interfejsu użytkownika odpowiada za wyświetlenie okna projekcyjnego i interakcję z użytkownikiem. Z punktu widzenia obserwatora projekcji, moduł ten zachowuje się jak zwykła aplikacja systemu *Windows*: otwiera główne okienko i udostępnia zestaw funkcji. Z technicznego punktu widzenia jest jednak zrealizowany w postaci dynamicznie wiązanej biblioteki (*dll*). Udostępnia on, na własnej liście eksportowej, funkcje pozwalające na otwarcie okna projekcyjnego, jego zamknięcie, zmianę tytułu oraz sprawdzenie, czy okno to jest otwarte. Funkcje te są wykorzystywane przez właściwą maszynę projekcyjną systemu *Daphnis*.

Moduł interfejsu użytkownika musi zapewnić obsługę okna projekcyjnego niezależną od głównego wątku programu poddawanego wizualizacji. Dlatego też pracuje on w ramach własnego wątku, współbieżnie do wątku głównego programu.

Interfejs programowy maszyny projekcyjnej obejmuje szereg funkcji, które powinny być wywoływane przez moduł interfejsu użytkownika (por. rozdział 7.4.1). Niektóre z nich są warunkiem poprawnego działania całego systemu. Odpowiadają one między innymi za kreślenie grafiki wewnątrz okienka projekcyjnego oraz taktowanie całej projekcji. Większość funkcji

jednak odpowiada wprost poszczególnym opcjom dostępnym w ramach interfejsu użytkownika, a nawet operacjom ustalającym stan określonych elementów (na przykład wyszarzeniu nieaktywnych opcji). Powoduje to, że implementacja modułu interfejsu użytkownika stała się zadaniem trywialnym (jeśli pominąć problemy z uruchomieniem głównego okna aplikacji z poziomu wątku uruchomionego w ramach biblioteki *dll*). Jedyną opcją w całości realizowaną w ramach modułu interfejsu użytkownika jest skalowanie i przesuwanie (przewijanie) obrazu.

7.7 SYSTEM *DAPHNIS* JAKO SYSTEM OTWARTY

Funkcjonalność systemu *Daphnis* można diametralnie poszerzyć uzupełniając go o nowe specjalizacje dwóch klas, mających zasadniczy udział w procesie sterowania projekcją, a w szczególności mających ogromny wpływ na bogactwo środków dostepnych dla wizualizatora. Klasy te to:

- Grele, odpowiedzialne za wytworzenie reprezentacji graficznej danych, oraz
- Translatory, sterujące przetwarzaniem wizualizowanych danych.

Obydwie wymienione wyżej klasy stanowią integralną część właściwej maszyny projekcyjnej systemu, i zostały bliżej opisane w rozdziale 7.3. Można też zauważyć, że są to jedyne klasy w obrębie maszyny, które posiadają specjalizacje (por. plansza 7.1).

Właśnie specjalizacje wymienionych wyżej klas mogą być wprowadzane przez użytkownika do systemu, czyniąc z *Daphnis* system otwarty. Użytkownik chcący poszerzyć system o nowe klasy obiektów powinien zaimplementować je w postaci składników *ActiveX*, zwanych też czasem składnikami *COM* [203, 205 – 208].

Składniki *ActiveX* stanowią rodzaj obiektowej biblioteki dynamicznej. Programy korzystające ze składników otrzymują obiektowo zorientowany interfejs programowy, pozwalający na powoływanie wcieleń – obiektów poszczególnych klas. Jest to zgodne z zasadą tak zwanego programowania składnikowego[9] (ang. *component programming*) [185].

System *Daphnis* narzuca na rozszerzające go składniki *ActiveX* następujące wymogi:

- Stworzona klasa musi implementować ściśle określony interfejs (odmienny dla greli i translatorów). Jest to wymóg charakterystyczny dla programowania składnikowego, w którym nie istnieje dziedziczenie w ścisłym sensie tego słowa.

[9] Innym znanym standardem umożliwiającym programowanie składnikowe jest CORBA [202].

- Stworzoną klasę należy zarejestrować w rejestrze systemowym *Windows* (ang. *Windows system registry*).

Po spełnieniu powyższych wymogów, dostarczone przez użytkownika klasy mogą być wykorzystywane na równi z klasami predefiniowanymi w obrębie systemu *Daphnis*. W szczególności nazwy klas greli i translatorów, a także atrybutów greli mogą być normalnie wykorzystywane w ramach skryptu konfiguracyjnego projekcji.

W poniższych podajemy dokładną specyfikację wymogów narzuconych przez system *Daphnis* na składniki *ActiveX* dostarczające dodatkowe klasy.

7.7.1 INTERFEJSY SKŁADNIKÓW ACTIVEX

Wszystkie klasy dołączane do systemu *Daphnis* w składnikach *ActiveX* muszą implementować interfejs o nazwie *IDaphnis*. Gwarantuje on możliwość dynamicznego tworzenia obiektów na podstawie danych pobranych ze skryptu, a także klonowania – tworzenia kopii obiektu. Oto specyfikacja interfejsu *IDaphnis*:

```
dispinterface IDaphnis
{
methods:
   [id(1)] boolean SetParamNum(double param);
   [id(2)] boolean SetParamNum(BSTR param);
   [id(3)] boolean Verify();
   [id(4)] IDispatch* Clone();
};
```

Metody *SetParamNum* i *SetParamStr* pozwalają na przekazanie do obiektu serii parametrów (liczbowych lub tekstowych) inicjalizujących ten obiekt zaraz po jego utworzeniu. Obiekt może oczekiwać przekazania określonych parametrów w określonej kolejności. Wartość zwracana przez metody określa, czy dany parametr został zaakceptowany jako poprawny. Po przekazaniu pełnej serii parametrów system *Daphnis* wywołuje metodę *Verify* celem sprawdzenia, czy cały proces inicjalizacji powiódł się. Metodę tę obiekt może też wykorzystać do sfinalizowania procesu inicjalizacji. Opisany sposób przekazywania parametrów nie precyzuje *a priori* ich liczby ani kolejności, otwierając tym samym drogę do tworzenia klas o przeciążonych listach parametrów – obiekt dynamicznie decyduje o zaakceptowaniu parametrów bądź też o ich odrzuceniu.

Metoda *Clone* powinna zwracać kopię wywoływanego obiektu.

Klasy reprezentujące translatory, oprócz interfejsu *IDaphnis*, muszą dodatkowo implementować interfejs *ITranslator* o następującej specyfikacji:

```
dispinterface ITranslator
{
methods:
   [id(1)] double Function(double x);
   [id(2)] boolean HasRevFunction();
   [id(3)] double RevFunction(double x);
};
```

Oto znaczenie poszczególnych metod:

- *Function* – dokonuje translacji dla wartości *x*.
- *HasRevFunction* – zwraca logiczną prawdę, jeśli klasa implementuje translację odwrotną.
- *RevFunction* – dokonuje odwrotnej translacji dla wartości *x*. Wartość metody nie jest określona, jeśli *HasRevFunction* nie zwraca logicznej prawdy.

Klasy reprezentujące grele muszą implementować interfejs *IGrel*:

```
dispinterface IGrel
{
methods:
   [id(1)] boolean Covers(long left, long top, long right, long bottom);
   [id(2)] boolean CoversXY(long x, long y);
   [id(3)] void OnPaint(long dc);
   [id(4)] long GetAttributeNum();
   [id(5)] BSTR GetAttributeName(short nAttrCode);
   [id(6)] double GetAttribute(short AttrCode);
   [id(7)] void SetAttribute(short AttrCode, double AttrVal);
};
```

Oto znaczenie poszczególnych metod:

- Metody *Covers* i *CoversXY* zwracają wartość określającą, czy zdefiniowany parametrami wywołania, odpowiednio, prostokąt lub punkt pokrywa się z obszarem grela.
- *OnPaint* wykreśla obraz grela w kontekście urządzenia zdefiniowanym przez parametr *dc*. Metoda ta hermetyzuje operacje graficzne zależne od platformy.
- *GetAttributeNum* zwraca liczbę atrybutów grela.
- *GetAttributeName* zwraca słowną nazwę atrybutu (na przykład "height").
- *GetAttribute, SetAttribute* – odpowiednio, zwraca lub ustawia wartość wskazanego atrybutu. Ustalenie wartości atrybutu nie powinno powodować przerysowania grela.

Należy zwrócić uwagę na fakt, że metody dotyczące atrybutów (*GetAttributeNum*, *GetAttributeName*, *GetAttribute*, *SetAttribute*) muszą uwzględniać także atrybuty pochodzące z ewentualnych klas nadrzędnych.

7.7.2 REJESTRACJA SKŁADNIKÓW ACTIVEX

Rejestracja składników *ActiveX* ma na celu umożliwienie systemowi *Daphnis* rozpoznanie i skojarzenie nazwy utworzonej przez użytkownika klasy (w brzmieniu używanym w skrypcie konfiguracyjnym) ze składnikiem *ActiveX*, w obrębie którego jest ona zdefiniowana. W tym celu należy stworzyć w rejestrze systemowym *Windows* klucz o nazwie identycznej z nazwą klasy, i skojarzyć z nim wartość ciągu identyfikatora klasy typu *GUID*. Klucz należy umieścić w folderze:

- *HKEY_CURRENT_USER\Software\JFrancik\Daphnis\Grels* w przypadku klasy greli,
- *HKEY_CURRENT_USER\Software\JFrancik\Daphnis\Translators* w przypadku definiowania klasy translatorów,

Dla przykładowej klasy greli o nazwie *SampleGrel* odpowiedni zapis rejestracyjny (w formacie programu *RegEdit*) może wyglądać następująco:

```
HKEY_CURRENT_USER\Software\JFrancik\Daphnis\Grels\SampleGrel =
                                {615BFBF4-C123-11D2-9B1F-444553540000}
```

Niezależnie od powyżej opisanej procedury, składniki powinny być niezależnie zarejestrowane w systemie *Windows* – zgodnie z regułami rejestrowania składników *ActiveX*.

Dla programistów używających środowiska *Microsoft Visual C++* przygotowywany jest element typu *Application Wizard*, ułatwiający i przyspieszający tworzenie składników *ActiveX* poszerzających funkcjonalność systemu *Daphnis*.

8
PRZYKŁADY DZIAŁANIA SYSTEMU DAPHNIS

Niniejszy rozdział zawiera opis kilku przykładowych animacji algorytmów, przygotowanych z pomocą systemu *Daphnis*. Większość przykładów ilustrujemy fragmentami tekstu źródłowego poddawanego wizualizacji programu, zawartością skryptu konfiguracyjnego oraz kolorowymi planszami przedstawiającymi sceny z projekcji. Zakładamy, że Czytelnik zna zasady przygotowywania programów do projekcji oraz tworzenia skryptów konfiguracyjnych. Zasady te przedstawiliśmy w rozdziale 6.4.

8.1 OPERACJA ZAMIANY DWÓCH ZMIENNYCH

Wprowadzając w rozdziale 4.2 trzy wymagania stawiane skutecznej wizualizacji algorytmów, odwoływaliśmy się do dość trywialnego przykładu wizualizacji zamiany dwóch zmiennych. Uwzględniając wszystkie trzy postulaty, wizualizacja operacji zamiany wartości zmiennych *A* i *B* powinna się sprowadzać do jednoczesnego ruchu elementu graficznego reprezentującego zmienną *A* w kierunku *B* i elementu graficznego reprezentującego zmienną *B* w kierunku *A*. Oto jak wygląda fragment programu realizujący taką operację, uzupełniony o odpowiednie wywołania systemu *Daphnis*:

```
double a, b;    Register("a", &a); Register("b", &b);
double tmp;     Register("tmp", &tmp);

a = 0.5;        Play(&a);              // inicjalizacja zmiennych
b = 1;          Play(&b);

tmp = b;        Play(&tmp, &b);        // zamiana wartości
b = a;          Play(&b, &a);
a = tmp;        Play(&a, &tmp);
```

A oto odpowiedni skrypt konfiguracyjny, korzystający z dwóch prostokątnych, barwnych prostokątów o zaokrąglonych rogach:

```
a is roundRectangle
{
   x = 200; y = 200;
   width = 100;
   height = linear(0, 1, 5, 150);
   redBrush = 255;
};

b is roundRectangle
{
   x = 500; y = 200;
   width = 100;
   height = linear(0, 1, 5, 150);
   greenBrush = 255;
};
```

Wysokość obydwu prostokątów jest uzależniona od wartości zmiennych.

Użytkownik chcący wprowadzić do projekcji szczegółowe informacje na temat wszystkich trzech operacji przypisania powinien jedynie zadeklarować w skrypcie konfiguracyjnym element graficzny reprezentujący zmienną pomocniczą:

```
tmp is roundRectangle
{
   x = 350; y = 400;
};
```

Powyższa deklaracja w zupełności wystarczy: pierwsza instrukcja przypisania (*tmp* = *b*) zniszczy zawartość zmiennej *tmp*, a zanim się to stanie, zmienna ta nie zostanie zainicjalizowana żadną wartością (zwartościowana) – zatem jej reprezentacja graficzna nigdy nie będzie widoczna. Ważna jest natomiast wartość atrybutów *x* i *y* – jako atrybuty przestrzenne zostaną one przejęte przez element reprezentujący wartość przepływającą do zmiennej *tmp* – w tym przypadku wartość zmiennej *b*. Widoczne to będzie jako ruch elementu reprezentującego zmienną *b* w kierunku pozycji o współrzędnych (*350, 400*) – zdefiniowanych przez aktora zmiennej *tmp*.

8.2 ALGORYTMY SORTOWANIA

Algorytmy sortowania stanowią klasyczny przedmiot zainteresowania animacji algorytmów. Przedstawiony poniżej skrypt konfiguracyjny pozwala stworzyć projekcję dowolnego algorytmu manipulującego elementami tablicy, i to tablicy o dowolnej długości:

```
Arr is roundRectangle
{
   index x = linear(0, 1, 30, 58);
   y = 120;
   width = 28;
   height = linear(0, 100, 50, 200);
   blueBrush = 128;
};
```

Na stronie 144 przytoczyliśmy fragment tekstu źródłowego programu implementującego algorytm **sortowania bąbelkowego**, uzupełniony o wywołania funkcji systemu *Daphnis*. Na planszy 8.1a przedstawiamy kadr z projekcji tego programu, zrealizowanej za pomocą przytoczonego powyżej skryptu. Projekcja taka nie uwzględnia jednak żadnych cech charakterystycznych dla tego właśnie algorytmu. Możliwe jest natomiast stworzenie skryptu przeznaczonego specjalnie do wizualizacji sortowania bąbelkowego. Kadr z takiej projekcji przedstawiamy na planszy 8.1b. Wykorzystujemy fakt, że elementy o indeksach mniejszych od *i* są zawsze posortowane (gdzie *i* – jedna ze zmiennych biorących udział w programie), i zaznaczamy te elementy kolorem zielonym. Dodatkowo wyróżniamy elementy poddawane w danej chwili operacji porównania (mają one indeksy równe *i* – *1* oraz *i*). Oto treść skryptu generującego taką projekcję:

```
Arr is roundRectangle
{
   index x = linear(0, 1, 30, 58);
   y = 200;
   width = 10;
   greenBrush = 128 + (index < #i);
   redBrush   = 128 + (index >= #i);
   flash redBrush = 128 * (index == #j || index == #j - 1);
};
i is rectangle
{
   x = linear(0, 1, 16, 44);
   y = 240;
   width = 5; height = 30;
   blueBrush = 240;
}
j is rectangle
{
   x = linear(0, 1, 16, 44);
   y = 240;
   width = 5; height = 30;
   greenBrush = 240;
};
```

Przedstawimy sposób przygotowania projekcji jeszcze jednego algorytmu sortującego, a mianowicie algorytmu **sortowania szybkiego** (ang. *quicksort*) [166]. Oto fragment tekstu źródłowego implementującego algorytm – którego wizualizację zamieszczamy na planszy 8.1c:

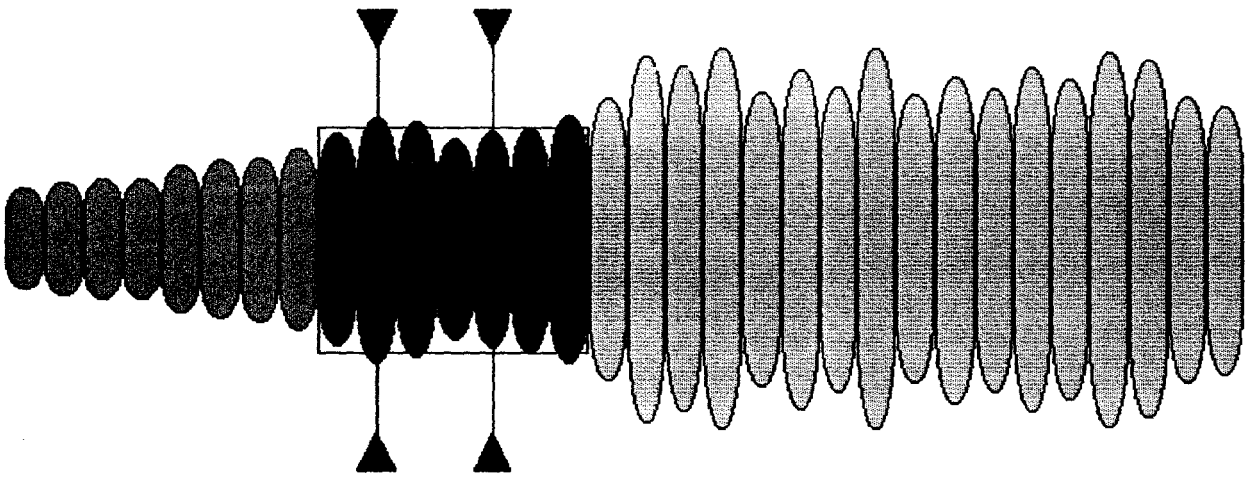
```
void Quicksort(int arr[], int p, int r)
{
   if (p < r)
   {
         int q = Partition(arr, p, r);
         Quicksort(arr, p, q);
         Quicksort(arr, q + 1, r);
   }
}

int Partition(int arr[], int p, int r)
{
                                  Register("p", &p); Play(&p);
                                  Register("r", &r); Play(&r);
   int x = arr[p];                Register("x", &x); Play(&x);
   int i = p - 1;                 Register("i", &i); Play(&i);
   int j = r + 1;                 Register("j", &j); Play(&j);
   while(1)
   {
         do { j--;                Play(&j); }
         while (arr[j] > x);
         do { i++;                Play(&i); }
         while (arr[i] < x);
         if (i < j)
         {
               int tmp = arr[i]; Register("tmp", &tmp); Play(&tmp, &arr[i]);
               arr[i] = arr[j];  Play(&arr[i], &arr[j]);
               arr[j] = tmp;     Play(&arr[j], &tmp);
                                 UnRegister(&tmp);
         }
         else
         {
                                  UnRegister(&p); UnRegister(&r);
                                  UnRegister(&x);
                                  UnRegister(&i); UnRegister(&j);
               return j;
         }
   }
}
```

Strategią stosowaną przez algorytm jest „dziel i rządź". Dla jej uwypuklenia zdecydowaliśmy się na wyróżnienie czerwonym kolorem tych elementów tablicy, które są sortowane w danym kroku algorytmu. Są to elementy o indeksach *p..r*. Pozostałe elementy są zaznaczone kolorem szarym, z tym, że elementy o indeksach mniejszych od *p* są wyświetlane ciemniejszym odcieniem szarości dla uwypuklenia faktu, że są one już posortowane. Wartości zmiennych indeksujących *i* i *j* zaznaczono specjalnym elementem graficznym, przydatnym do zaznaczania pozycji w tablicach. Kluczowym zagadnieniem przy wizualizacji sortowania

a)

b)



c)

Plansza 8.1 Wizualizacje algorytmów sortowania przygotowane za pomocą systemu *Daphnis*.
a) Sortowanie bąbelkowe, prosta wizualizacja. b) Sortowanie bąbelkowe, wizualizacja zaawansowana.
c) Sortowanie szybkie.

szybkiego jest sposób przedstawienia mediany *x*. Wykorzystaliśmy element graficzny o kształcie ramki, której wysokość odpowiada wartości mediany. Ramka ta pokrywa całą szerokość aktualnie sortowanego podciągu. Dzięki temu użytkownik otrzymuje prosty, wizualny klucz rozdziału elementów w kolejnych krokach algorytmu: elementy na tyle małe, iż mieszczą się w obrębie ramki mają znaleźć się po lewej stronie, podczas gdy elementy nie mieszczące się w ramce przesuwane są w prawo. Poniżej zamieszczamy odpowiedni skrypt konfiguracyjny tej wizualizacji:

```
Arr is roundRectangle
{
   index x = linear(0, 1, 30, 50);
   y = 120;
   width = 20;
   height = linear(0, 100, 50, 200);
   redBrush = (#index < p) ? 128 : (index <= r) ? 255 : 192;
   greenBrush = (#index < p) ? 128 : (index <= r) ? 0 : 192;
   blueBrush = (#index < p) ? 128 : (index <= r) ? 0 : 192;
};
i is arrayPointer
{
   value x = linear(0, 1, 16, 44);
   y = 120;
   width = 20; height = 240;
   greenBrush = 255;
}
j is arrayPointer
{
   value x = linear(0, 1, 16, 44);
   y = 120;
   width = 20; height = 240;
   greenBrush = 255;
}
x is frame
{
   value x = linear(0, 1, 16, 44) of (#p + #r) / 2;
   y = 120;
   width = linear(0, 1, 16, 44) of (#r - #p);
   height = linear(0, 100, 50, 200);
   blueBrush = 255;
};
```

Zwróćmy uwagę na fakt, że zmienne *p* i *r*, wyznaczające granicę aktualnie rozpatrywanego podciągu, nie posiadają własnego aktora: są pozbawione indywidualnej reprezentacji graficznej. Ich wartości są jednak bardzo dobrze odzwierciedlone za pośrednictwem innych elementów: poprzez kolorystykę elementów tablicy oraz szerokość ramki reprezentującej medianę.

8.3 SYMULACJA AUTOMATU KOMÓRKOWEGO

Symulację automatu komórkowego zaprezentujemy na przykładzie gry w życie Conwaya [172 – 175]. Zasady gry opisaliśmy na stronie 17. Oto uzupełniony tekst programu źródłowego implementującego grę w życie:

```
const int N = 30;
BOOL a[2][N][N];
Register("plansza", a, N * N * 2);

// inicjalizacja planszy
...

int g = 0;      // g - indeks jednej z dwóch plansz
while (kontynuacja())
{
   for (i = 0; i < N; i++)
         for (j = 0; j < N; j++)
         {
               int count =
                     a[g][(i+N-1)%N][(j+N-1)%N] +
                     a[g][(i+N-1)%N][j] +
                     a[g][(i+N-1)%N][(j+1)%N] +
                     a[g][i][(j+N-1)%N] +
                     a[g][i][(j+1)%N] +
                     a[g][(i+1)%N][(j+N-1)%N] +
                     a[g][(i+1)%N][j] +
                     a[g][(i+1)%N][(j+1)%N];
               BOOL bLive = a[g][i][j];
               bLive = (bLive && (count == 2 || count == 3))
                     || (!bLive && count == 3);

               a[!g][i][j] = bLive;          Play(&a[!g][i][j]);
         }
   g = !g;
}
```

Za pomocą systemu *Daphnis* dokonamy wizualizacji zawartości tablicy *a*, reprezentującej planszę gry. Nie ma potrzeby przedstawienia jakichkolwiek innych zmiennych. Implementacja komputerowa gry w życie wymaga zastosowania dwóch plansz (*a[0]* i *a[1]*), przechowujących dane dotyczące dwóch kolejnych generacji automatu. Zamieszczony poniżej skrypt wyświetla obydwie plansze obok siebie, pozwalając użytkownikowi śledzić dwie kolejne generacje automatu:

```
plansza is rectangle3D
{
   index x = (linear(0, 1, 50, 60) of index % 30) + 350 * (index / 900);
   index y = linear(0, 1, 50, 60) of (index % 900) / 30;
   width = 10; height = 10;
   redBrush = x ? 255 : 192;
```

```
    greenBrush = x ? 0 : 192;
    blueBrush = x ? 0 : 192;
};
```

System *Daphnis* nie obsługuje w sposób jawny tablic wielowymiarowych, stąd nieco skomplikowany sposób przeliczenia indeksu tablicy *a*, traktowanej jak tablica jednowymiarowa. Stan komórki kodowany jest kolorami: komórki żywe są czerwone, martwe zaś – szare. Przykładowy kadr z projekcji przedstawia plansza 8.2.

Zaproponowane podejście, w którym pokazuje się dwie kolejne generacje, różni się od najczęściej spotykanych realizacji gry w życie. Aby wyświetlać każdorazowo tylko jedną planszę, wystarczy umieścić reprezentację graficzną obydwu planszy i określić, która z nich ma być widoczna. Wymaga to niewielkiej modyfikacji skryptu:

```
plansza is rectangle3D
{
   index x = linear(0, 1, 50, 60) of index % 30;
   visible = index div 900 == #g;
   // pozostałe pozycje bez zmian
```

Aby powyższa konstrukcja zadziałała, trzeba jeszcze uzupełnić tekst źródłowy programu tak, by śledzona była wartość zmiennej *g*.

8.4 ALGORYTM TRANSLACJI WYRAŻEŃ NA ODWROTNĄ NOTACJĘ POLSKĄ

Zaprezentujemy obecnie wizualizację algorytmu tłumaczenia wyrażeń arytmetycznych z postaci infiksowej na odwrotną notację polską (plansza 8.3). W górnej części okienka projekcyjnego wyświetlana jest zawartość ciągu wejściowego. Kolejno analizowane znaki są przenoszone bądź to do ciągu wyjściowego, położonego w dolnej części okienka, bądź też na umieszczony po prawej stronie stos operatorów. Ilustracja przedstawia moment, w którym kolejnym analizowanym elementem ciągu wejściowego jest nawias zamykający; operatory umieszczone na szczycie stosu zostają przeniesione do ciągu wyjściowego.

```
char *pIn = in;                    Register("in", pIn, N_IN);
char *pOut = out;                  Register("out", pOut, N_OUT);
char *pStack = stack;              Register("stack", pStack, N_STACK);

*pStack = '(';                                Play(pStack);
do
{
   if (isalpha(*pIn))
         { *pOut++ = *pIn;                    Play(pOut-1, pIn); }
```

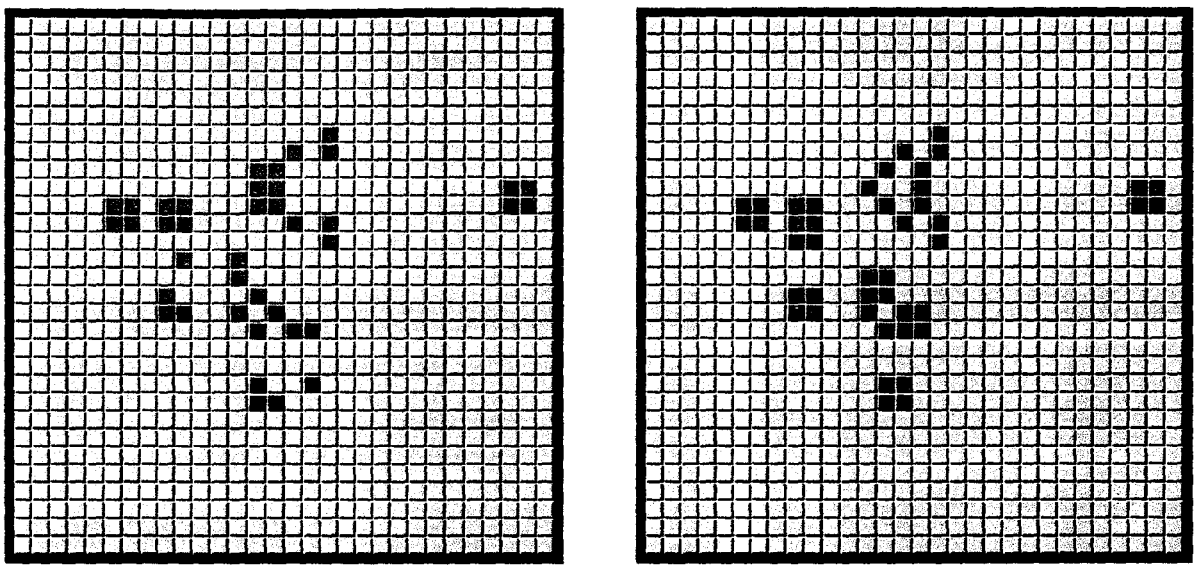
```
    else
    if (isoperator(*pIn))
    {
          while (pr(*pStack) >= pr(*pIn))
                { *pOut++ = *pStack--;        Play(pOut-1, pStack+1);
          *++pStack = *pIn;                   Play(pStack, pIn);
    }
    else
    if (*pIn == '(')
          { *++pStack = *pIn;                 Play(pStack, pIn); }
    else
    if (*pIn == ')' || *pIn == '\0')
    {
          while (*pStack != '(')
                { *pOut++ = *pStack--;        Play(pOut-1, pStack+1);
          *pStack--;                          Play(pStack+1);
    }
    else
          break;      // illegal character
} while (*pIn++);
*pOut = '\0';                                 Play(pOut);
```

W tekście powyższego programu wielokrotnie powtarzają się operatory z działaniami ubocznymi: ++ oraz −−. Ponieważ argumentami tych operatorów są wskaźniki na struktury danych, które są poddawane wizualizacji, skomplikowało to w wielu przypadkach wywołanie funkcji *Play*. Aby uzyskać dostęp do właściwych danych, konieczne było wprowadzenie w obrębie zapisu parametrów aktualnych wyrażeń realizujących operacje odwrotne do tych, które wykonały wspomniane operatory.

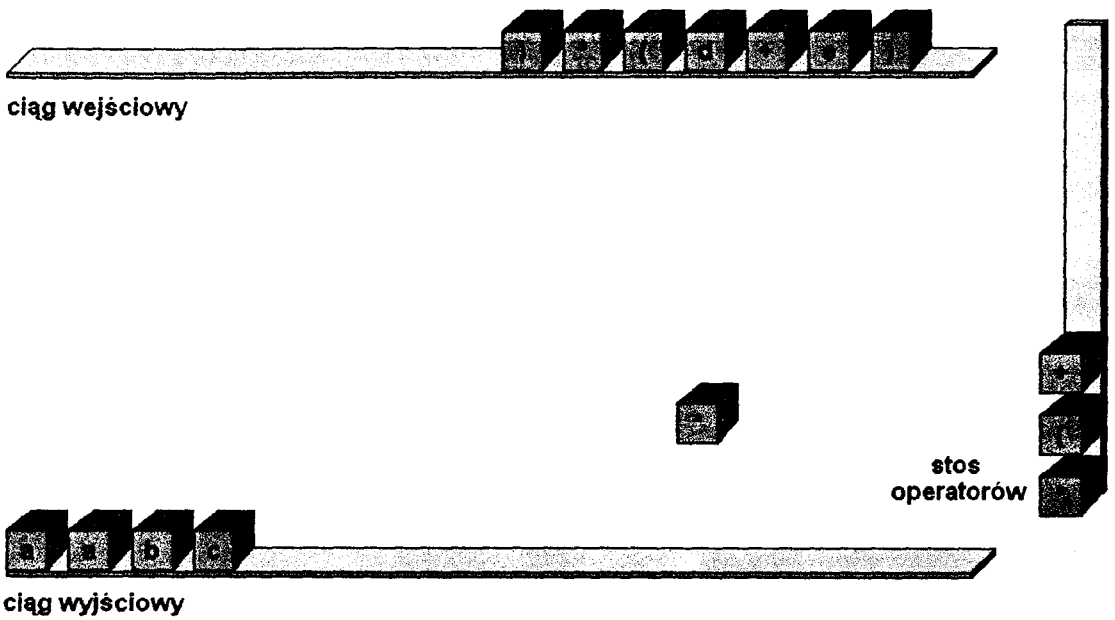
Skrypt konfiguracyjny do omawianej projekcji nie jest skomplikowany:

```
noncopy in is charBitmap("cube3d.bmp")
{
   index x = linear(0, 1, 30, 70);
   index y = 50;
   char = value;
};
noncopy out is charBitmap("cube3d.bmp")
{
   index x = linear(0, 1, 30, 70);
   index y = 380;
   char = value;
};
noncopy stack is charBitmap("cube3d.bmp")
{
   index x = 700;
   index y = linear(0, 1, 340, 300);
   char = value;
};
```

W definicjach wszystkich trzech biorących udział w projekcji aktorów użyto specyfikatora *noncopy*. W ten sposób akcentujemy fakt, że w prezentowanym algorytmie dane – zarówno



Plansza 8.2. Wizualizacja gry w życie przygotowana za pomocą systemu *Daphnis*



Plansza 8.3. Wizualizacja algorytmu translacji wyrażeń z postaci infiksowej na odwrotną notację polską, uzyskana za pomocą systemu *Daphnis*

w ciągu wejściowym, jak i na stosie operatorów – są odczytywane tylko raz, a po odczycie stają się nieistotne. Zastosowanie specyfikatora *noncopy* powoduje, że elementy graficzne reprezentujące odczytywane zmienne są nie kopiowane, a przesuwane na odpowiednie miejsca docelowe. Jak widać na planszy 8.3, w ciągu wejściowym (w górnej części obrazu) pozostały puste miejsca po odczytanych elementach.

Do utworzenia elementów graficznych (greli) klasy *charBitmap* wykorzystano plik z mapą bitową, dzięki czemu wygląd elementarnych „cegiełek" tej projekcji można było łatwo zaprojektować za pomocą dowolnego rastrowego edytora graficznego. W podobny sposób stworzone zostało tło obrazu przedstawiające prowadnice – metaforę tablic.

8.5 ALGORYTM DOSTĘPU DO TABLICY Z UŻYCIEM FUNKCJI MIESZAJĄCEJ

Na przykładzie algorytmu dostępu do tablicy z użyciem funkcji mieszającej przedstawimy trzy różne podejścia do wizualizacji tego samego algorytmu.

Plansza 8.4 przedstawia kadr z projekcji zrealizowanej na poziomie szczegółowym. Użyto stosunkowo niewielkiego zbioru danych wejściowych (tablica mieszająca liczy tylko 23 elementy). W lewej części okna pojawiają się słowa wczytywane z pliku i przeznaczone do umieszczenia w tablicy. Niewielkie kółko wskazuje indeks tablicy wyznaczony dla danego słowa przez funkcję mieszającą, a strzałka – pozycję, pod którą słowo faktycznie jest zapamiętywane, z uwzględnieniem ewentualnych kolizji. Tego rodzaju wizualizacja jest niewątpliwie przydatna do celów dydaktycznych, może też być użyta na wstępnym etapie uruchamiania algorytmu. W wielu przypadkach wymaga się jednak przedstawienia działania algorytmu pracującego ze znacznie większym zbiorem danych wejściowych. Na planszy 8.5 przedstawiono wizualizacje tego samego algorytmu, przy czym rozmiar tablicy wynosi 501 elementów. Siłą rzeczy wizualizacja staje się mniej szczegółowa, za to może lepiej przedstawić syntetyczny obraz zachowania się algorytmu. I tak, plansza 8.5a przedstawia histogram obrazujący liczbę odwołań, które wystąpiły podczas prób dostępu do poszczególnych elementów tablicy, plansza 6b zaś pokazuje wykres średniej liczby konfliktów w funkcji współczynnika wypełnienia tablicy. Histogram podobny do zobrazowanego na planszy 8.5a, stworzony dla projekcji z małym zbiorem danych wejściowych, znajduje się w prawej części planszy 8.4. Całościowy, albo syntetyczny ogląd zachowania algorytmu pozwala dostrzec wiele aspektów niemożliwych do zaobserwowania w przypadku wizualizacji szczegółowej. I tak na

planszy 8.5a możemy prześledzić efekty tzw. klasteryzacji, czyli liczniejszego gromadzenia się kolizji w wybranych zakresach elementów tablicy. Możemy też wykryć i wskazać elementy danych szczególnie często się powtarzające w ciągu wejściowym. Z kolei analizując efekty projekcji pokazanej na planszy 8.5b możemy ustalić maksymalny poziom współczynnika wypełnienia tablicy, przy którym dostęp jest zadowalająco efektywny.

Z technicznego punktu widzenia, zobrazowanie zarówno histogramu liczby odwołań, jak i wykresu średniej liczby konfliktów wymagało ręcznego zaprogramowania odpowiednich struktur danych w obrębie poddawanego wizualizacji programu. Możliwe jest także opracowanie odpowiednich translatorów i zastosowanie właściwego skryptu tak, by poprawki w tekście źródłowym nie były konieczne, a wszystkie potrzebne obliczenia (w tym zbieranie danych w postaci histogramu) było realizowane po stronie systemu *Daphnis* – wydaje się to jednak rozwiązaniem zbyt skomplikowanym.

8.6 WIEŻE HANOI

Wieże Hanoi, sławna, klasyczna układanka stanowiąca pretekst wielu komputerowo realizowanych demonstracji graficznych, da się rozwiązać za pomocą bardzo krótkiego, rekurencyjnego algorytmu:

```
void hanoi(int a, int c, int b, int n)
{
   if (n > 0)
   {
         hanoi(a, b, c, n - 1);
         move(a, c);
         hanoi(b, c, a, n - 1);
   }
}
```

Wywołanie funkcji *move* reprezentuje przełożenie krążka z pałeczki *a* na pałeczkę *c*. Funkcja ta zwykle zawiera implementację graficzną tej operacji. Choć dla wygenerowania sekwencji przełożeń krążków nie jest potrzebna żadna dodatkowa struktura danych, to jednak – dla potrzeb wizualizacji – wygodnie będzie reprezentować układ trzech pałeczek z nanizanymi na nie krążkami w postaci dwuwymiarowej tablicy. Elementy tablicy *Disks* określają szerokość krążka nanizanego na odpowiedniej pozycji (wysokości) pałeczki. Stanowią one dogodny przedmiot wizuazliacji dla systemu *Daphnis*. Dodatkowo, wprowadzamy tablicę *pos* określającą liczbę krążków nanizanych na poszczególne pałeczki. Przy tych założeniach, uzupełniony o wywołanie systemowe *Daphnis* tekst funkcji *move* wygląda następująco:

planszy 8.5a możemy prześledzić efekty tzw. klasteryzacji, czyli liczniejszego gromadzenia się kolizji w wybranych zakresach elementów tablicy. Możemy też wykryć i wskazać elementy danych szczególnie często się powtarzające w ciągu wejściowym. Z kolei analizując efekty projekcji pokazanej na planszy 8.5b możemy ustalić maksymalny poziom współczynnika wypełnienia tablicy, przy którym dostęp jest zadowalająco efektywny.

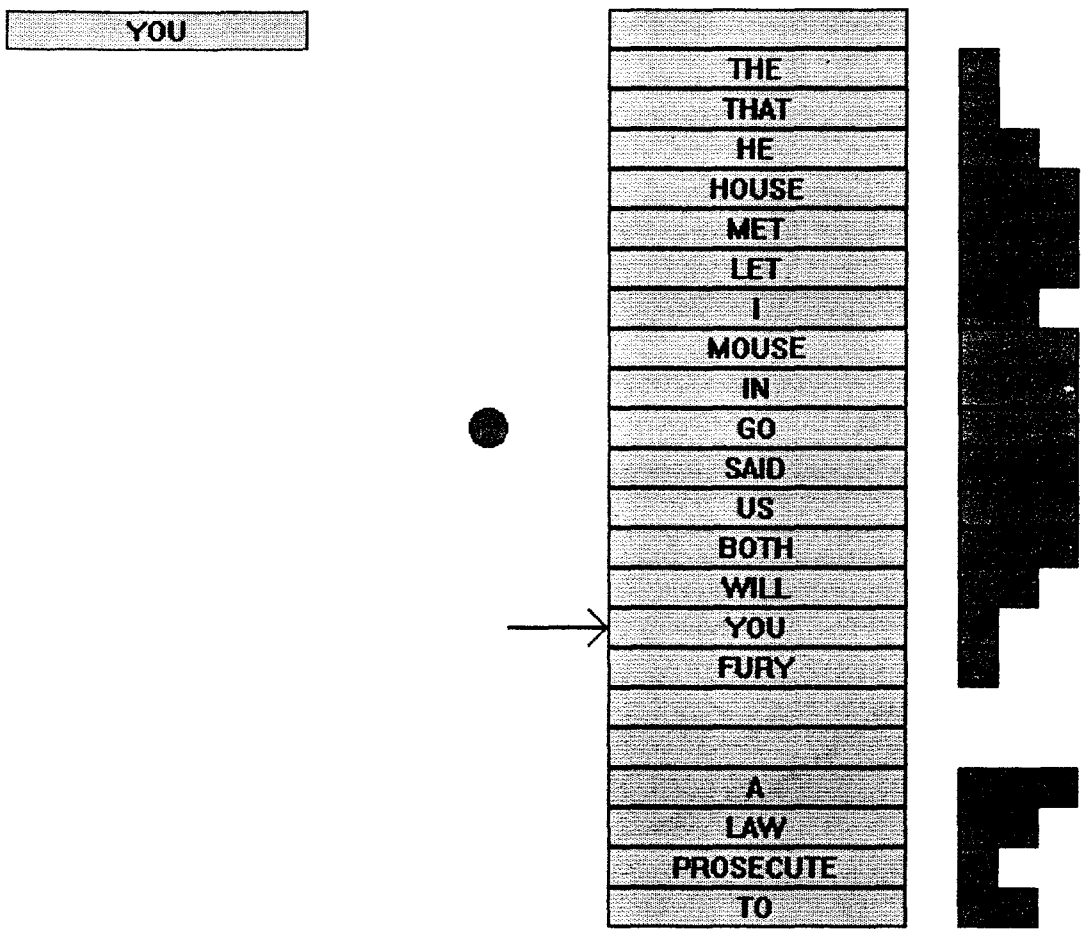
Z technicznego punktu widzenia, zobrazowanie zarówno histogramu liczby odwołań, jak i wykresu średniej liczby konfliktów wymagało ręcznego zaprogramowania odpowiednich struktur danych w obrębie poddawanego wizualizacji programu. Możliwe jest także opracowanie odpowiednich translatorów i zastosowanie właściwego skryptu tak, by poprawki w tekście źródłowym nie były konieczne, a wszystkie potrzebne obliczenia (w tym zbieranie danych w postaci histogramu) było realizowane po stronie systemu *Daphnis* – wydaje się to jednak rozwiązaniem zbyt skomplikowanym.

8.6 WIEŻE HANOI

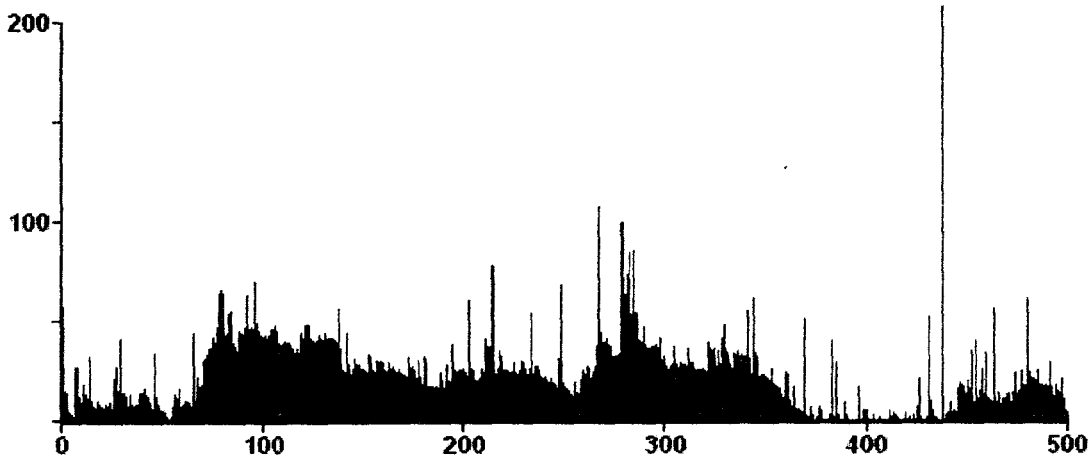
Wieże Hanoi, sławna, klasyczna układanka stanowiąca pretekst wielu komputerowo realizowanych demonstracji graficznych, da się rozwiązać za pomocą bardzo krótkiego, rekurencyjnego algorytmu:

```
void hanoi(int a, int c, int b, int n)
{
   if (n > 0)
   {
         hanoi(a, b, c, n - 1);
         move(a, c);
         hanoi(b, c, a, n - 1);
   }
}
```

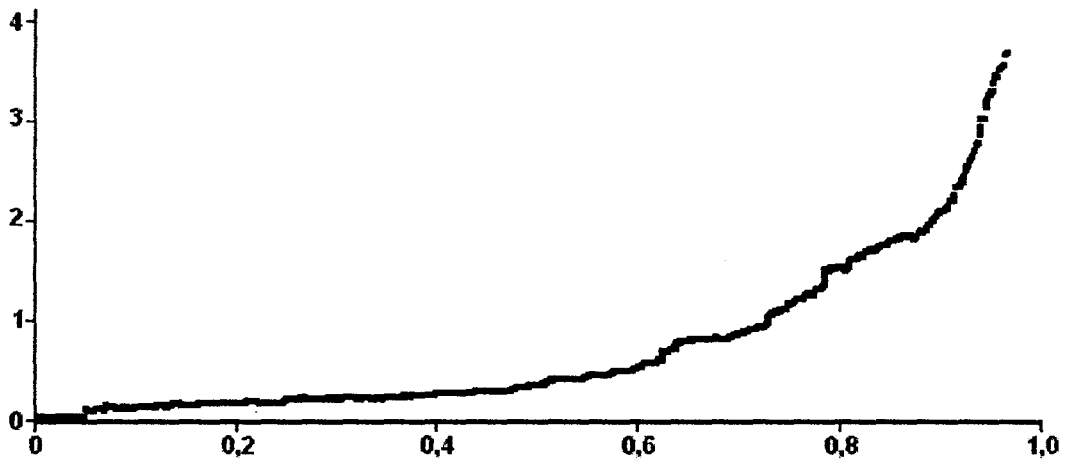
Wywołanie funkcji *move* reprezentuje przełożenie krążka z pałeczki *a* na pałeczkę *c*. Funkcja ta zwykle zawiera implementację graficzną tej operacji. Choć dla wygenerowania sekwencji przełożeń krążków nie jest potrzebna żadna dodatkowa struktura danych, to jednak – dla potrzeb wizualizacji – wygodnie będzie reprezentować układ trzech pałeczek z nanizanymi na nie krążkami w postaci dwuwymiarowej tablicy. Elementy tablicy *Disks* określają szerokość krążka nanizanego na odpowiedniej pozycji (wysokości) pałeczki. Stanowią one dogodny przedmiot wizuazliacji dla systemu *Daphnis*. Dodatkowo, wprowadzamy tablicę *pos* określającą liczbę krążków nanizanych na poszczególne pałeczki. Przy tych założeniach, uzupełniony o wywołanie systemowe *Daphnis* tekst funkcji *move* wygląda następująco:



Plansza 8.4. Wizualizacja algorytmu dostępu do tablicy przy pomocy funkcji mieszającej – podejście szczegółowe, dla małej próbki danych wejściowych. Wizualizację zrealizowano z wykorzystaniem systemu *Daphnis*.



a)



b)

Plansza 8.5. Wizualizacja algorytmu dostępu do tablicy przy pomocy funkcji mieszającej – podejście syntetyczne dla dużej próbki danych wejściowych. a) Histogram obrazujący liczbę odwołań, które wystąpiły podczas prób dostępu do poszczególnych elementów tablicy. b) Wykres średniej liczby konfliktów w funkcji współczynnika wypełnienia tablicy. Obydwie wizualizacje zrealizowano przy pomocy systemu *Daphnis*.

```
void move(int a, int b)
{
   Disks[b][++pos[b]] = Disks[a][pos[a]--];
   Play(&Disks[b][pos[b]], &Disks[a][pos[a]+1]);
}
```

A oto skrypt konfiguracyjny:

```
noncopy disks is roundRectangle
{
   straight x = linear(0, 1, 120, 340) of index / 12;
   straight y = linear(0, 1, 280, 260) of index % 12;
   width = linear(0, 12, 0, 200);
   height = 20;
   redBrush = random(0, 1);
   greenBrush = random(0, 1);
   blueBrush = random(0, 1);
};
base is rectangle
{
   index x = linear(0, 1, 120, 340); y = 295;
   width = 200; height = 10;
};
stick is rectangle
{
   index x = linear(0, 1, 120, 340); y = 165;
   width = 2; height = 250;
};
```

Specyfikator *noncopy* aktora *disks* zapewnia usunięcie krążka po przełożeniu go na inną pałeczkę. Aktorzy *base* i *stick* to ozdobniki: na generowanym obrazie odpowiadają odpowiednio podstawom wież i pałeczkom.

Poddawany wizualizacji program poszerzymy o globalne definicje tablic i inicjalizującą je funkcję:

```
int Disks[3][12];
int pos[3];

void RunHanoi(int n)
{                                  Register("base", pos, 3);
                                   Register("stick", pos, 3);
                                   Register("disks", Disks[0], 3*n);
   for (int i = 0; i < n; i++)
         Disks[0][i] = n - i;
                                   Play(Disks, n);
   pos[0] = n - 1;
   pos[1] = -1;
   pos[2] = -1;                    Play(pos, 3);
   hanoi(0, 2, 1, n);
}
```

Tablicę *pos* wykorzystano dodatkowo jako pretekst do utworzenia elementów dekoracji *base* i *stick*. Tak zdefiniowana wizualizacja ma jednak pewną wadę: mimo, że na ekranie widnieje obraz trzech pałeczek, krążki przemieszczają się pomiędzy nimi najkrótszą drogą, bez uwzględnienia faktu, że są na te pałeczki nanizane. Wprowadzimy więc punkty wyznaczające tor ruchu krążków, i umieścimy je ponad pałeczkami. W tym celu dopiszemy do skryptu dodatkowego aktora:

```
noncopy top is rectangle
{
   index x = linear(0, 1, 120, 340); y = 30;
};
```

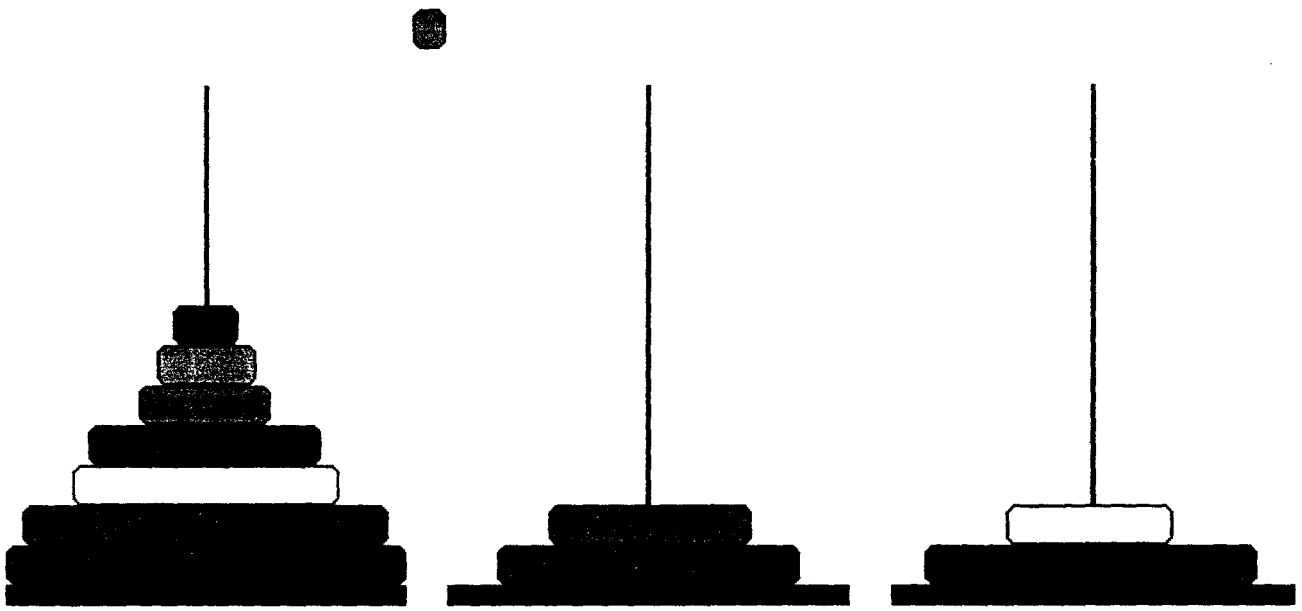
Wprowadzimy też dodatkową tablicę pomocniczą:

```
int top[3];
void RunHanoi(int n)
{                                    Register("top", top, 3);
   // pozostała część funkcji pozostaje bez zmian
   ....
}
```

Wreszcie podzielimy ruch elementów graficznych na trzy etapy, uwzględniające punkty pośrednie umiejscowione ponad pałeczkami. W tym celu funkcję *move* zapiszemy w następujący sposób:

```
void move(int a, int b)
{
   top[a] = Disks[a][pos[a]--];  Play(&top[a], &Disks[a][pos[a]+1]);
   top[b] = top[a];              Play(&top[b], &top[a]);
   Disks[b][++pos[b]] = top[b];  Play(&Disks[b][pos[b]], &top[b]);
}
```

Efekt wizualizacji przedstawia plansza 8.6.



Plansza 8.6. Animacja „wież Hanoi" stworzona za pomocą systemu *Daphnis*

9
UWAGI KOŃCOWE

Zanim przejdziemy do podsumowania wyników prac, przypomnijmy raz jeszcze tezę pracy:

> Przy zastosowaniu metody animacji algorytmów opartej na śledzeniu przepływu danych możliwe jest automatyczne wprowadzenie do wizualizacji oprogramowania elementów znacząco zwiększających poziom abstrakcji tej wizualizacji

W ramach pracy zaproponowaliśmy trzy wymagania stawiane wizualizacji oprogramowania cechującej się wysokim poziomem abstrakcji (por. rozdział 4.2). Są to:

- **Postulat obrazowania przepływu danych**: wizualizacja powinna wykorzystywać płynną animację w celu przedstawienia operacji przepływu danych, jakie mają miejsce podczas procesu wykonania algorytmu.
- **Postulat przestrzennej redukcji informacji**: wizualizacja powinna wykluczyć z projekcji informacje dotyczące nieistotnych zmiennych, zachowując przy tym całościowy obraz przepływu danych w algorytmie.
- **Postulat temporalnej redukcji informacji**: wizualizacja powinna zrównoleglić wykonanie niektórych przedstawianych operacji elementarnych.

Zaproponowana **metoda animacji algorytmów** oparta na **śledzeniu przepływu danych** gwarantuje spełnienie wyżej wymienionych wymagań bez potrzeby wykonania jakichkolwiek działań ze strony użytkownika poza tymi, które są niezbędne do uruchomienia podstawowej projekcji. Samo zatem wprowadzenie do wizualizacji elementów zwiększających poziom abstrakcji następuje w sposób automatyczny. Tym samym teza pracy została wykazana.

Algorytm animacji algorytmów oparty na śledzeniu przepływu danych wyprowadziliśmy w rozdziale 4. Skorzystaliśmy w tym celu z formalizmu sieci Petriego. Jednak praca, której tezę udowodnilibyśmy jedynie teoretycznie, ograniczając się do sformułowania algorytmu,

nie byłaby pracą pełną. Dlatego też zaimplementowaliśmy system animacji algorytmów *Daphnis*, który wykorzystuje efekty rozważań teoretycznych. Zastosowano w nim algorytm trójkolorowy, 1,5-krokowy – będący jedną z najbardziej zaawansowanych odmian metody opartej na śledzeniu danych. Opisowi systemu poświęciliśmy rozdziały 5 – 7. W rozdziale ósmym przedstawiliśmy wiele przykładów obrazujących zrealizowane z pomocą *Daphnis* projekcje.

Implementacja systemu *Daphnis* jest praktycznym dowodem słuszności tezy pracy. Tworzenie projekcji przy użyciu systemu jest łatwe i gwarantuje automatyczne spełnienie trzech wymienionych postulatów – bez konieczności podejmowania przez użytkownika jakichkolwiek specjalnych działań. Zauważmy, że ich spełnienie w istniejących dotąd systemach nie było możliwe bez podjęcia w tym celu świadomego wysiłku przez projektantów poszczególnych wizualizacji.

Analiza działania systemu *Daphnis* pozwoliła nie tylko potwierdzić efekty prac teoretycznych, ale także je uzupełnić. Doświadczenia uzyskane podczas uruchamiania pierwszych projekcji pozwoliły wprowadzić poprawki ograniczające moc postulatu temporalnej redukcji informacji (por. str. 100). W trakcie obserwacji pracy systemu uznaliśmy także, że implementacja algorytmu wizualizacji opartej na śledzeniu danych w wersji jeszcze bardziej wyrafinowanej, niż ta, którą zastosowaliśmy w systemie (na przykład algorytmu wielokrokowego – por. str. 105), nie byłaby zasadna – mogłaby utrudnić użytkownikowi zrozumienie wizualizowanego algorytmu.

Projekt i implementacja systemu stały się też okazją do zaproponowania **obiektowej architektury systemu**. Tworzy ona łańcuch przekształceń, przez który przechodzą dane pobrane z poddawanego wizualizacji programu zanim zostaną przedstawione w postaci graficznej. Architektura ta, wykorzystująca koncepcję modelu *POST*, stwarza możliwość konfigurowania przekształceń danych za pomocą tekstowego pliku skryptowego. Dzięki temu system *Daphnis* pozwala na przygotowywanie wizualizacji o bardzo wysokim stopniu abstrakcji, charakteryzujących się znaczną zawartością intencyjną, w sposób prawie całkowicie **deklaratywny**. W dotychczasowych rozwiązaniach, projekcje o takim stopniu komplikacji przekształceń danych wymagały na ogół zaszycia instrukcji przekształcających dane wprost w tekście źródłowym programu.

Oryginalnym osiągnięciem pracy była też realizacja **sposobu połączenia** systemu wizualizacji z poddawanym wizualizacji programem. Dzięki zastosowaniu dołączanej dynamicznie biblioteki (*dynamic link library, dll*) system umożliwia animację programów wyrażonych

w niemal dowolnym języku dostępnym w środowisku *Windows*. Jak dotąd, podjęto owocne próby uruchomienia wizualizacji programów przygotowanych w *C++* oraz w języku *Visual Basic*.

Za istotne dokonanie należy też uznać **otwartą architekturę** systemu *Daphnis*. Użytkownik może poszerzać funkcjonalność systemu tworząc i rejestrując nowe składniki oprogramowania, które mają bezpośredni wpływ na różnorodność środków, jakimi dysponuje wizualizator. Zastosowano w tym celu technikę programowania składnikowego, opartą na szeroko przyjętym standardzie *COM/ActiveX*.

Naszym zdaniem system *Daphnis* korzystnie wyróżnia się na tle innych dostępnych rozwiązań – przede wszystkim pod względem łatwości przygotowania nowych projekcji. Za najpilniejsze stojące przed nami zadanie niewątpliwie uznać należy jak najszersze rozpropagowanie systemu *Daphnis*. Może się on stać użytecznym narzędziem przede wszystkim dla nauczycieli informatyki, ale także i dla programistów – szczególnie po zakończeniu przewidzianych prac nad modułami wspomagania pracy wizualizatora. Mamy nadzieję, że zaproponowane rozwiązania przyczynią się do upowszechnienia wizualizacji oprogramowania.

W najbliższym czasie system *Daphnis* zostanie wdrożony do procesu dydaktycznego na Politechnice Śląskiej. System jest też dostępny w sieci Internet[10]. W ubiegłych latach poprzednik systemu *Daphnis*, *WinSANAL* był kilkukrotnie prezentowany w ramach ekspozycji Politechniki Śląskiej na imprezach targowych, w tym na targach CeBIT w Hanowerze. Niestety, sytuacja finansowa Uczelni stawia przyszłość tego rodzaju promocji systemu pod znakiem zapytania.

System *Daphnis* jest oprogramowaniem gotowym do użytku, nie jest jednak jeszcze projektem zamkniętym – wręcz przeciwnie, należy go traktować jako zaawansowany prototyp. Za szczególnie obiecujące kierunki przyszłych badań i rozwoju systemu uznać należy:

- Poszerzenie systemu o środki wizualizacji struktur dynamicznych, takich, jak listy i drzewa. Obecnie możliwości systemu *Daphnis* są w tym zakresie dość ograniczone.
- Wprowadzenie widoku tekstu źródłowego programu, z zaznaczeniem przepływu sterowania.
- Wprowadzenie możliwości interaktywnej zmiany wartości danych metodą bezpośredniej manipulacji – istniała ona w jednej z wersji systemu *WinSANAL* [109].

[10] Pod adresem http://www-zo.iinf.polsl.gliwice.pl/ ~jfrancik/aa/download.htm.

- Poszerzenie systemu o możliwość tworzenia wizualizacji trójwymiarowych. Obecnie tworzone projekcje co najwyżej korzystają z elementów graficznych o przestrzennym wyglądzie (por. planszę 8.3).
- Realizacja badań nad udźwiękowieniem projekcji. Pierwsze, próbne dźwiękowe wizualizacje algorytmów sortowania już zostały przy pomocy systemu *Daphnis* zrealizowane.
- Wprowadzenie możliwości realizacji projekcji rozproszonych w środowisku sieciowym. Prace w tym kierunku były już z naszym udziałem prowadzone przy wykorzystaniu systemu *SANAL* [103]).
- Dodanie do sytemu modułów wspomagających pracę wizualizatora, w szczególności preprocesora tekstów źródłowych programów oraz edytora graficznego do tworzenia skryptów konfiguracyjnych. Prace implementacyjne nad tymi modułami zostały już rozpoczęte.
- Poszerzenie zestawu gotowych do wykorzystania projekcji. Większość z istniejących obecnie stworzono dla celów testowych oraz demonstracyjnych. Ich przegląd można znaleźć w rozdziale 8. Podczas wdrażania systemu, między innymi w ramach prac dydaktycznych, spodziewać się należy szybkiego napływu nowo tworzonych projekcji.

Spośród interesujących kierunków przyszłych prac, które wymagają dalszych rozważań teoretycznych, wymienimy:

- Wykorzystanie statycznej analizy struktury programu do wzbogacenia projekcji. Informacje pozyskane w wyniku takiej analizy mogą być istotnym uzupełnieniem informacji dotyczących przepływu danych zbieranych w sposób dynamiczny, podczas wykonywania wizualizowanego programu. Elementy analizy statycznej zostały już wykorzystane w systemie *Daphnis* celem ograniczenia temporalnej redukcji informacji podczas wizualizacji pętli.
- Poszukiwanie właściwego modelu wizualizacji dla nieimperatywnych aspektów oprogramowania, w tym dla języków deklaratywnych i architektur obiektowych. Określenie możliwości i granic wykorzystania zaproponowanej metody wizualizacji.
- Opracowanie metod wizualizacji bardzo dużych systemów.

BIBLIOGRAFIA

WIZUALIZACJA OPROGRAMOWANIA

Spośród publikacji przedstawiających ogólne zagadnienia związane z wizualizacją oprogramowania, nie tylko najobszerniejsza ale i najnowsza jest monografia [11]. Dość obszerny jest także artykuł przeglądowy [5]. Poza wymienionymi niżej, warto też sięgnąć do pozycji książkowej [74], klasycznego artykułu [73], a także do publikacji [105, 107, 116].

1. M.H.BROWN. *Perspectives on algorithm animation.* Proc. Of ACM SIGCHI'88 Conf. on Human Factors in Computing Systems, Washington D.C., USA, 1988, str. 33-38
2. A. HYRSKYKARI. *Development of program visualization systems.* Technical Report A-1995-3, University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland, 1995.
3. B. A. MYERS. *Taxonomies of Visual Programming and Program Visualisation.* Journal of Visual Languages and Computing, Vol. 1, 1990, str. 97-123.
4. M. PETRE, B.A. PRICE. *Why Computer Interfaces Are Not Like Paintings: the User as a Deliberate Reader.* Proc. of East-West International Conference on Human-Computer Interaction, Moskwa, 1992.
5. A. PRICE, R. M. BAECKER, I. S. SMALL. *A Principled Taxonomy of software Visualisation.* Journal of Visual Languages and Computing, Vol. 4, 1993, str. 211-266.
6. S. P. REISS. *Program Visualization: Where We Go From Here.* IFIP Transactions A Vol: A-12, 1992, str. 218-27.
7. S. P. REISS. *Generating abstractions for visualization.* Report CS-92-35, Dept. of Computer Science, Brown University, 1992.
8. G. C. ROMAN, K. C. COX. *A Taxonomy of Program Visualization Systems.* IEEE Computer, no 12, 1993, str. 11-24.
9. G. C. ROMAN, K. C. COX. *Program visualization: the art of mapping programs to pictures.* International Conference on Software Engineering, New York, 1992, str. 412-420.
10. G. C. ROMAN, K. C. COX. *Abstraction in algorithm animation.* Proceedings of 1992 IEEE Workshop on Visual Languages, Los Alamitos, CA, USA, 1992, str. 18-24.
11. J.T. STASKO, J.B. DOMINGUE, M.H. BROWN, B.A. PRICE, editors. *Software Visualisation. Programming as a Multimedia Experience.* Foreword by Jim Foley. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1998. ISBN 0-262-19395-7.

12. J. T. STASKO, C. PATTERSON. *Understanding and characterising software visualization systems.* Proceedings of IEEE Workshop on Visual Languages, Seattle, WA, USA, 1992, str. 3-10.
13. *Arne's Bibliography on Software Visualization.* Witryna WWW. http://i44s11.info.uni-karlsruhe.de/~frick/SoftVis/bibliography.html

Model pojmowania algorytmów i programów przez człowieka

14. M.E. CROSBY, J. STELOVSKY. *How Do We Read Algorithms?* IEEE Computer No 1, 1990, str. 24-35.
15. V. FIX, S. WIEDENBECK, J. SCHOLTZ. *Mental representations of programs by novices and experts.* Proceedings of Inter CHI'93, Addison-Wesley, 1993, str. 74-79.
16. L. FORD. *How programmers visualise programs.* 5th Workshop on Empirical Studies of Programmers, 1993.
17. V. RADIYA. *A Model of Human Approach to Describing Algorithms using Diagrams.* Proc. of IEEE Workshop on Visual Languages, Seattle, WA, USA, 1992, str. 261-4.

Zaawansowane techniki graficzne i dźwiękowe

Artykuł [19] jest klasyczną pozycją w dziedzinie. Dostępna jest wersja z kolorowymi ilustracjami oraz taśma video. Problem udźwiękowienia wizualizacji porusza też artykuł poświęcony systemowi *LOGOMEDIA* [134].

18. M. H. BROWN, M. A. NAJORK. *Algorithm Animations Using 3D Interactive Graphics.* Proc. ACM Symposium on User Interface Software and Technology, 1993, str. 93-100.
19. M. H. BROWN, J. HERSHBERGER. *Color and Sound in Algorithm Animation.* Computer, Vol. 25, N°12, 1991, str. 52-63. Ukazał się również jako SRC Research Report 110a, z kolorowymi ilustracjami, oraz Report 110b, wraz z taśmą video.
20. J. M. FRANCIONI, L. ALBRIGHT, J. A. JACKSON. *Debugging parallel programs using sound.* SIGPLAN Notices, Vol. 26, N°12, 1992, str. 68-75.
21. H. KOIKE. *The role of another spatial dimension in software visualization.* ACM Transactions on Information Systems, 11(3), 1993, str. 266-286.
22. H. LIEBERMAN. *A Three-Dimensional Representation For Program Execution.* Proceedings of the 1989 IEEE Workshop on Visual Languages. IEEE Computer Society Press, New York, 1989, str. 111-116.
23. J. T. STASKO, J. F. WEHRLI. *Three-Dimensional Computation Visualization.* Proceedings of IEEE/CS Symposium of Visual Languages, Bergen, Norway, 1993.

Techniki typograficzne

Warto sięgnąć po artykuły poświęcone systemowi *SEE* [64, 65], zawierające interesującą analizę problemu.

24. R. J. MIARA, J.A. MUSSELMAN, J.A. NAVARRO, B. SHNEIDERMAN. *Program Indentation and Comprehensibility.* Communications of the ACM, 26(11), 1983, str. 861-867.
25. F. NORCIO. *Indentation, Documentation, and Programmer Comprehension.* Proc. of Human Factors in Computing Systems CHI `82, New York, ACM Press, 1982, str. 118-120.

Reżyseria projekcji i sterowanie parametrami projekcji

26. P. SZMAL. *Sterowanie parametrami projekcji w systemach wizualizacji algorytmów.* Załącznik do raportu z Badań Własnych BW 420/Rau2/96 na temat *Wizualizacja obliczeń w nie dedykowanych językach programowania*, Instytut Informatyki Politechniki Śląskiej, Gliwice, 1996.

Wizualizacja dużych zestawów danych
Niezwykle interesujące przykłady zastosowań tego typu wizualizacji można znaleźć w [138, 139].

27. D. JERDING, J.T. STASKO. *The Information Mural: A Technique for Displaying and Navigating Large Information Spaces.* IEEE Transactions on Visualisation and Computer Graphics, Vol. 4, No. 3, July-Sept 1998, str. 257-271.
28. J. MUTHUKUMARASAMY, J.T. STASKO. *Visualising Program Executions on Large Data Sets Using Semantic Zooming.* Technical Report GIT-GVU-95-02, Georgia Institute of Technology, Atlanta, GA, USA, 1995.

Wizualizacja struktur drzewiastych

29. K. DOBOSZ. *Wizualizacja grafowych struktur danych.* Artykuł przyjęty do druku w Zeszytach Naukowych Politechniki Śląskiej, Gliwice, 1999.
30. J. T. STASKO, C. R. TURNER. *Tidy Animations of Tree Algorithms.* Proc. of the 1992 IEEE Workshop on Visual Languages, Seattle, WA, USA, September 1992. str. 216-218.

Model POST
Wykorzystywany w systemie *Daphnis* model POST, stworzony przez M. Mostefaia, został też użyty w systemie *Siamoa* [116, 120].

31. M. MOSTEFAI. *Un Modèle d'Architecture oriente objet pour la conception de plates-formes de simulation interactive.* Rozprawa doktorska, Université des Sciences et Technologies de Lille, Francja, 1994.
32. M. MOSTEFAI. *P.Os.T. : An architectural model for Interactive Simulation Interfaces Design.* IEEE/SMC'93 Conference, Systems Engineering in the service of the Humans, le Touquet, Francja, Vol. 2, 1993, str. 550-555.

Bezpośrednia manipulacja elementami graficznymi
Warto też sięgnąć do pozycji dotyczących systemów wizualizacji oprogramowania, w których zastosowano technikę bezpośredniej manipulacji elementami graficznymi: Whizz [101, 102] i Tango [79].

33. L. CARDELLI. *Building user interfaces by direct manipulation.* Proc. of the ACM SIGGRAPH Symp. on User Interface Software, Banff, Alberta, Canada, 1988, str. 152-166.
34. L. CYPHER. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.
35. T. TONOUCHI, K. NAKAYAMA, S. MATSUOKA, S. KAWAI. *Creating visual objects by direct manipulation.* Proc. of IEEE Workshop on Visual Languages, Seattle, WA, 1992, str. 95-101.

WIZUALIZACJA NAUKOWA I PRZEMYSŁOWA ORAZ PROGRAMOWANIE WIZUALNE

Wizualizacja naukowa

36. K.W.BRODLIE (RED.). *Scientific Visualization. Techniques and Applications.* Springer-Verlag, Heidelberg, 1992.

Wizualizacja przemysłowa

37. R. CUPEK. *Wizualizacja systemów automatycznego sterowania.* Zeszyty Naukowe Pol. Śl., z.23, Gliwice 1993.
38. R. CUPEK. *Wizualizacja rozproszonych procesów przemysłowych.* Rozprawa doktorska, Instytut Informatyki Politechniki Śląskiej, Gliwice, 1998.

Programowanie wizualne

Problematykę tę poruszono też w [3], a także w publikacjach poświęconych systemowi *Siamoa* [116, 118].

39. M. BURNETT, A. GOLDBERG, LEWIS. *Visual Object-Oriented Programming*. Prentice-Hall, 1995.
40. L. DAVIS, R. M. KELLER. *Data Flow Program Graphs*. IEEE Computer No 2, 1982, str. 26-41.
41. M. HIRAKAWA, T. ICHIKAWA. *Advances in visual programming*. ICSI '92. Proceedings of the Second International Conference on Systems Integration, 1992, str. 538-43.
42. J. POSWIG, G. VRANKAR, C. MORAGA. *Interactive animation of visual program execution*. Proceedings of IEEE Symposium on Visual Languages, Bergen, Norway, 1993, str. 180-7.
43. J. RASURE, M. YOUNG. *Data flow visual languages*. IEEE Potentials, Vol. 11 Iss. 2, 1992, str. 30-3.

ZASTOSOWANIA WIZUALIZACJI OPROGRAMOWANIA

44. L. AMBLER, M. M. BURNETT. *Influence of visual technology on the evolution of language environments* Computer, Vol. 22, N°10, 1989, str. 9-22.
45. M.H. BROWN. *The 1993 SRC Algorithm Animation Festival*. SRC Research Report 126, 1994.
46. M.H. BROWN. *The 1992 SRC Algorithm Animation Festival*. Proceedings of IEEE Symposium on Visual Languages, Bergen, Norway, 1993, str. 116-23.
47. M.H. BROWN, R. SEDGEWICK. *Techniques for Algorithm Animation*. IEEE Software, 2(1), 1985, str. 28-39.
48. M.H. BROWN, R. SEDGEWICK. *The Electronic Classroom: a Progress Report*. Video tape, 1984.
49. M.H. BROWN, R. SEDGEWICK. *Progress Report: Brown University Instructional Computing Laboratory*. ACM SIGCSE Bulletin, 16(1), 1984, str. 91-101.
50. M.H. BROWN, N. MEYROWITS, A. VAN DAM. *Personal Computer Networks and Graphical Animation: Rationale and Practice for Educatiorn*. ACM SIGCSE Bulletin 15(1), 1983, str. 296-307.
51. W.H. HUGGINS, D.R. ENTWISLE. *Computer Animation for the Academic Community*. Proceedings of 1969 Spring Joint Computer Conference, 1969, str. 623-627.
52. C. KEHOE, J.T. STASKO. *Using Animations to Learn about Algorithms: An Ethnographic Case Study*. Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-96-20, 1996.
53. A.W. LAWRENCE, A.N. BADRE, J.T. STASKO. *Empirically Evaluating the Use of Animations to Teach Algorithms*. Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO, USA, 1994, str. 48-54.
54. S. MUKHERJEA, J.T. STASKO. *Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger*. ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3, September 1994, str. 215-244.
55. S. MUKHERJEA, J.T. STASKO. *Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding*. Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, 1993, str. 456-465.

56. J.T. STASKO. *Supporting Student-Built Algorithm Animation as a Pedagogical Tool (Formal Demonstration).* Proceedings of the ACM SIGCHI '97 Conference on Human Factors in Computing Systems – Extended Abstracts, Atlanta, GA, 1997, str. 24-25.

57. J. T. STASKO. *Using Student-Built Algorithm Animations as Learning Aids.* Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, CA, 1997, str. 25-29.

58. J.T. STASKO, A. BADRE, C. LEWIS. *Do Algorithm Animations Assist Learning? An Empirical Study and Analysis.* Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, Amsterdam, Netherlands, 1993, str. 61-66.

SYSTEMY WIZUALIZACJI OPROGRAMOWANIA

Dobry przegląd licznych systemów wizualizacji algorytmów można znaleźć w [5, 8, 11]. Poświęcona przede wszystkim systemowi *BALSA* książka [74] również zawiera opis wielu innych rozwiązań. Klasyfikację istniejących systemów można znaleźć w [3, 5, 8, 12]. Poniżej zamieściliśmy literaturę odnoszącą się do poszczególnych systemów, w kolejności, w jakiej opisane one zostały w rozdziale 3. niniejszej pracy.

Filmy animowane

59. R. M. BAECKER. *Sorting Out Sorting.* Morgan Kaufmann, Los Altos, California. Dźwiękowa, kolorowa taśma video, 30 minutowa, zaprezentowana na ACM SIGGRAPH '81 i przyjęta do ACM SIGGRAPH Video Review N°7, 1983.7

60. K.S. BOOTH. *PQ Trees.* 16 mm kolorowy film, 12 minutowy, 1975.

61. F.R.A. HOPGOOD. *Computer Animation Used as a Tool in Teaching Computer Science.* Proceedings of 1974 IFIP Congress, 1974, str. 889-892.

62. K. C. KNOWLTON. *L[6]: Bell Telephone Laboratories Low-Level Linked List Language.* 16 mm czarno-biały, dźwiękowy film, 16 minutowy. Technical Information Libraries, Bell Laboratories, Inc., Murray Hill, NJ, 1966

63. K. C. KNOWLTON. *L[6]: Part II. An Example of L[6] Programming.* 16 mm black and white sound film, 16 mm czarno-biały, dźwiękowy film, 30 minutowy. Technical Information Libraries, Bell Laboratories, Inc., Murray Hill, NJ, 1966

Systemy do ładnego drukowania programów

64. R. M. BAECKER, A. MARCUS. *Human factors and Typography for More Readable Programs.* Addison-Wesley, Reading, 1990

65. R.M. BAECKER, A. MARCUS. *Design Principles for the Enhanced Presentation of Computer Program Source Text.* Proceedings of Human Factors in Computing Systems (CHI `88), ACMPress, New York, NY, USA, 1986, str. 51-58.

66. P. BORRAS, D. CLÉMENT, TH. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG, V. PASCUAL. *Centaur: the system.* Proc. Of 3rd ACM SIGSOFT Symposium on Software Development Environments, Bostom, 1988.

67. I. JACOBS, L. RIDEAU. *A Centaur Tutorial.* Institut National de Recherche en Informatique et en Automatique INRIA, Rapport Technique N° 141, Sophia Antipolis, Francja, 1992.

68. A.-M. DERY, L. RIDEAU. *Distibuted Architecture for Programming Environments.* Institut National de Recherche en Informatique et en Automatique INRIA, Rapport Technique N° 2918, Sophia Antipolis, Francja, 1996.

69. D. E. KNUTH. *Literate Programming.* The Computer Journal,27(2), 1984, str. 97-111.

Incense, Amethist i Pascal Genie

70. B.A. MYERS, R. CHANDHOK, A. SAREEN. *Automatic Data Visualization for Novice Pascal Programmers.* Proceedings of The IEEE Workshop on Visual Languages, IEEE ComputerSociety Press, New York, NY, USA, 1988, str. 192-198.

71. B. A. MYERS. *Incense: A System for Displaying Data Structures.* Computer Graphics, Vol. 17, N°3, 1983, str. 115-125.

Microsoft Visual C++

72. *Microsoft® Visual C++®.* Zintegrowane środowisko programowania. Microsoft Corporation. Version 6.0, 1998.

Balsa i Zeus

73. M. H. BROWN. *Exploring Algorithms Using Balsa II.* IEEE Computer, Vol. 21, N°5, 1988, str. 14-36.

74. M.H. BROWN. *Algorithm Animation.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988. ISBN 0-262-02278-8.

75. M. H. BROWN, R. SEDGEWICK. *A System for Algorithm Animation.* Proceedings of ACM SIGGRAPH '84, ACM Press, New York, 1984, str. 177-186.

76. M. H. BROWN. *Zeus: A System for Algorithm Animation and Multi-View Editing.* Proceedings of IEEE Workshop on Visual Languages. Kobe, Japon, 1991, str. 4-9.

77. *Algorithm Animation at SRC (Zeus).* SRC. Witryna WWW. http://www.research.digital.com/SRC/zeus/home.html

78. *Zeus-based animations of distributed algorithms and communication protocols.* Witryna WWW. http://ls4-www.informatik.uni-dortmund.de/RVS/zada.html

Tango, XTango i Polka

79. J. T. STASKO. *Using Direct Manipulation to Build Algorithm Animations by Demonstration.* Proceedings of the ACM SIGCHI'91 Conference on Human Factors in Computing Systems , New Orleans, LA, USA, May 1991, str. 307-314.

80. J. T. STASKO. *Tango: A Framework and System for Algorithm Animation.* IEEE Computer, Vol. 23, No. 9, 1990, str. 27-39.

81. J.T. STASKO. *The path-transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces.* Journal of Visual Languages and Computing, Vol. 1, No. 3, 1990, str. 213-236.

82. J. T. STASKO. *TANGO: A Framework and System for Algorithm Animation.* Rozprawa doktorska. Brown University, Department of Computer Science, Providence , RI 02912, N° CS-89-30, 1989.

83. J. STASKO. *Animating algorithms with XTANGO.* SIGACT News, Vol. 23, Iss. 2, 1992, str. 67-71.

84. J. T. STASKO. *Polka animation designer's package.* Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, 1992.

85. *Software Visualisation Group.* Graphics, Visualization & Usability Center, Georgia Tech. Witryna WWW. http://www.cc.gatech.edu/gvu/softviz/SoftViz.html

ANIM

86. J. L. BENTLEY, B. W. KERNIGHAN. *A System for Algorithm Animation.* Computing Systems, Vol. 4, N°1, 1991, str. 5-30.

87. J. L. BENTLEY, B. W. KERNIGHAN. *ANIM.* Kolekcja programów w ANSI C używanych do wizualizacji, dostępna dla komputerów pracujących w systemie UNIX pod adresem ftp://ftp.att.com/netlib/research. AT&T Bell Laboratories, Murray Hill, NJ, 1992.

Animus

88. R. A. DUISBERG. *Visual programming of program visualisations. A gestural interface for animating algorithms.* IEEE Computer Society Workshop on Visual Languages, Linkoping, Sweden, 1987, str. 55-66.

89. R. A. DUISBERG. *Animated graphical interfaces using temporal contraints.* Proceedings of the ACM SIGGHI'86 Conference on Human Factors in Computing Systems, Boston, MA, 1986, str. 131-136.

90. R. A. DUISBERG. *Contraint Based Animation: Temporal Contraints in the Animus System.* Rozprawa doktorska, University of Washington, Seatle, WA, USA, 1986. Również dostępna jako: Technical Report 86-09-01.

91. R. L. LONDON, R. A. DUISBERG. *Animating Programs using Smalltalk.* Computer, Vol. 18, N°8, 1985, str. 61-71.

UWPI

92. R.R. HENRY, K.M. WHALEY, B. FORSTALL. *The University of Washington Illustrating Compiler.* Proceedings of The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, New York, 1990, str. 223-233.

Pavane

93. D. HART, E. KRAEMER, G.-C. ROMAN. *Interactive Visual Exploration of Distributed Computations.* Washington University, Department of Computer Science, St. Louis, MO, USA, 1997.

94. G.-C ROMAN, K. C. COX, C. D. WILCOX, J. Y. PLUN. *Pavane: a System for Declarative Visualization of Concurrent Computations.* Journal of Visual Languages and Computing, Vol. 3, N°2, 1992, str. 161-193.

95. K. COX, G. C. ROMAN. *Visualizing Concurrent Computations.* Proceedings of the IEEE Workshop on Visual Languages, 1991, str. 18-24.

96. G.-C. ROMAN, H.C. CUNNINGHAM. *Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency.* IEEE Transactions on Software Engineering, 16, 1990, str. 1361-1373.

97. G.-C. ROMAN, K.C. COX. *A Declarative Approach to Visualizing Concurrent Computations.* Computer, 22(10), 1989, str. 25-36.

98. *Pavane.* Washington University. Witryna WWW. http://swarm.cs.wustl.edu/pavane.html

99. *Concurrent Systems Group.* Washington University. Witryna WWW. http://swarm.cs.wustl.edu/index.html

Whizz

100. S. CHATTY. *La construction d'interfaces Homme-machine animées.* Rozprawa doktorska, Centre d'Orsay, Universite de Paris-Sud, 1992.
101. S. CHATTY. *Defining the dynamic behaviour of animated interfaces.*
102. M. BEAUDOUIN-LAFON, Y. BERTEAUD, S. CHATTY. *Creating direct manipulation interfaces with* X_{TV}. 1st European Conference on XWindow System, 1990.

SANAL

103. J. FRANCIK, P. SZMAL. *Wielostanowiskowa wizualizacja algorytmów w sieciach komputerowych.* Seminarium Sieci Komputerowe, Zeszyty Naukowe Politechniki Śląskiej, seria Informatyka, 30, 1996, str. 293-310.
104. J. FRANCIK, P. SZMAL. *Wizualizacja algorytmów z wykorzystaniem systemu SANAL.* X Konferencja Informatyka w Szkole, Torun, 1994.
105. J. FRANCIK, P. SZMAL. *Wizualizacja synchroniczna algorytmów – realizacja w systemie SANAL.* Zeszyty Naukowe Politechniki Śląskiej, seria Informatyka 27, 1994, str. 37-54.
106. J. FRANCIK. *System animacji algorytmów SANAL z możliwością realizacji projekcji quasi-równoległych (wersja jednostanowiskowa).* Praca dyplomowa magisterska, Instytut Informatyki, Wydział Automatyki, Elektroniki i Informatyki Politechniki Śląskiej, Gliwice, 1993.

WinSANAL

107. P. SZMAL, J. FRANCIK. *Algorithm animation and debugging with the WinSanal system.* Proc. of IASTED Conference Applied Informatics, Innsbruck, Austria, 1997.
108. P. SZMAL, J. FRANCIK. *WinSanal: System animacji algorytmów dla potrzeb dydaktyki i inzynierii programowania.* Informatyka 7-8/1997.
109. P. SZMAL, J. FRANCIK, M. NOWAK. *Systemy wizualizacji algorytmów wspomagajace badania naukowe.* III krajowa konferencja Komputery w badaniach naukowych KOWBAN, Polanica Zdrój, 1996, str. 317-322.
110. P. SZMAL, J. FRANCIK. *WinSanal: System animacji algorytmów dla potrzeb dydaktyki i inzynierii programowania.* Druga Konferencja Informatyka na Wyższych Uczelniach dla Gospodarki Narodowej, Gdańsk, 1996, str. 49-54.
111. M. NOWAK. *Mechanizmy prezentacji danych w postaci symbolicznej i interakcyjnego wprowadzania danych dla systemu animacji algorytmów Sanal.* Praca dyplomowa magisterska, Instytut Informatyki, Wydział Automatyki, Elektroniki i Informatyki Politechniki Śląskiej, Gliwice, 1996.
112. P. SZMAL, K. DOBOSZ, J. FRANCIK, P. GONERA I A. MIGAS. *Metody wizualizacji sterowanej danymi w środowisku Smalltalk-80.* Raport końcowy z realizowanych badań statutowych. Instytut Informatyki Politechniki Śląskiej, Gliwice 1995.
113. P. SZMAL, J. FRANCIK, A. TOMASZEWSKA, K. WÓJCIK. *Wizualizacja dynamicznych struktur danych w języku Smalltalk-80.* Raport końcowy z realizowanych badań statutowych. Instytut Informatyki Politechniki Śląskiej, Gliwice 1994.
114. *Algorithm Visualization and Animation - mapping programs to pictures.* Witryna WWW. Instytut Informatyki Politechniki Śląskiej, Gliwice, http://www-zo.iinf.polsl.gliwice.pl/~jfrancik/aa

Siamoa

115. J. FRANCIK, P. SZMAL, F. VAN DE VEIRE. *Siamoa – A System for Visual Programming, Program Visualisation and Debugging*. Proc. of Advanced Visual Interfaces AVI '98, ACM Press, L'Aquila, Włochy, 1998, str. 289-291.
116. F. VAN DE VEIRE. *La Simulation et l'Animation Modulaire d'Algorithmes en Langage Objet*. Rozprawa doktorska. Université des Sciences et Technologies, Lille, Francja, 1997.
117. F. VAN DE VEIRE. *Modularly Animated Algorithms*. IMACS Multiconference IEEE/SMC Symposium on Modelling, Analysis and Simulation CESA'96, Lille, Francja, Vol. 2, 1996, str. 670-675.
118. F. VAN DE VEIRE. *Program Visualization and Visual Programming* European Simulation Multiconference Modelling and Simulation ESM'96, Budapeszt, 1996.
119. F. VAN DE VEIRE. *An Interactive and Convivial Object Oriented Simulation for Animation Algorithms*. European Simulation Multiconference Modelling and Simulation ESM'95, Praga, 1995, str. 599-603.
120. M. MOSTEFAI, F. VAN DE VEIRE. *An Object Oriented Architectural Model for Animating Algorithms* Proceedings of the European Simulation Symposium, ESS'94, Vol. 1, Istanbul, Turcja, 1994, str. 107-111.
121. *L'animation d'algorithmes*. Laboratorium I^3D, Uniwersytet w Lille, Francja. Witryna WWW. http://w3-cal.univ-lille1.fr/~fv/index.htm

Pozostałe systemy wizualizacji dla języków imperatywnych

122. J. DETREVILLE. *The GraphVBT interface for programming algorithm animations*. Proc. of 1993 IEEE Symposium on Visual Languages, Los Alamitos, CA, 1993, str. 26-31.
123. P.A. GLOOR. *AACE-algorithm animation for computer science education*. IEEE Workshop on Visual Languages, Seattle, WA, USA, 1992, str. 25-31.
124. P.A. GLOOR, S. DYNES, I. LEE. *Animated algorithms – a hypermedia learning environment for introduction to algorithms*. Wydanie na CD-ROM dla komputerów Apple MacIntosh, 1993.
125. E. HELTULLA, A. HYSKYKARI, K.-J. RÄIHÄ. *Graphical Specification of Algorithm Animations with ALLADIN*. Proc. of The Twenty-Second Annual Hawaii International Conference on System Science. IEEE Computer Society, New York, 1989, str. 892-901.
126. D.T. LEE, S.-M. SHEU, C.-F. SHEN. *GeoSheet: A Distributed Visualisation Tool for Geometric Algorithms*, 1994.
127. M. MORICONO, D. F. HARE. *Visualizing Program Designs Through Pegasys*. IEEE Computer, Vol. 18, N°8, 1985, str. 72-85.
128. S. P. REISS. *Interacting with the FIELD Environment*. Software Practice and Experience, Vol. 20, (S-1), 1989.
129. S. P. REISS. *Pecan: Program Development Systems that support Multiple Views*. IEEE Transaction on Software Engineering, Vol. SE-11, N°3, 1985, str. 276-285.
130. W. REYNOLDS. *Computer animation with Scripts and Actors*. Computer graphics, Vol. 16, N°3, 1982, str. 289-296.
131. *The Complete Collection of Algorithm Animations – 43 sites*. Witryna WWW. http://www.cs.hope.edu/~alganim/ccaa/ccaa.html
132. *World-wide algorithm animation*. Witryna WWW. http://cuiwww.unige.ch/eao/www/WWW94/paper.html

Systemy wizualizacji dla języków funkcyjnych

133. H. BOCKER, G. FISCHER, H. NIEPER. *The Enhacement of Understanding through Visual Representations.* Proceedings of the Computer Human Interaction'86 Conference, Human Factors in Computing Sustems - III, 1986, str. 44-50.
134. J. DIGIANO, R. M. BAECKER. *Program Auralization: Sound Enhancements to the Programming Environment.* Proc. of Graphics Interface'92, Palo Alto, 1992, str. 44-53.

Systemy wizualizacji dla języków deklaratywnych

135. M. BRAYSHAW, M. EISENSTADT. *A Practical Graphical Tracer for Prolog. International Journal of Man-Machine Studies*, 35(5), 1991, str. 597-631.
136. M. EISENSTADT, M. BRAYSHAW. *The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming.* Journal ofLogic Programming, 5(4), 1988, str. 1-66.

Systemy wizualizacji dla języków obiektowych

137. W. DE PAUW, R. HELM, D. KIMELMAN, J. VLISSIDES. *Visualizing the behavior of object-oriented systems.* SIGPLAN Notices, Vol. 28 Iss. 10, 1993, str. 326-37.
138. D. F. JERDING, J.T. STASKO. *Visualizing Message Patterns in Object-Oriented Program Executions.* Technical Report GIT-GVU-96-15, Georgia Institute of Technology, Graphics, Visualization and Usability Center, College of Computing, Atlanta, GA, 1994.
139. D. F. JERDING, J. T. STASKO. *Using visualization to foster object-oriented program understanding.* Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Graphics, Visualization and Usability Center, College of Computing, Atlanta, GA, USA, 1994.
140. C. LAFFRA, A. MALHOTRA. *HotWired – a Visual Debugger for C++.* Proceedings of the USENIX 6th C++ Technical Conference, 1994.
141. D.B. LANGE, Y. NAKAMURA. *Program Explorer: A Program Visualiser for C++.* Proceedings of USENIX Conference on Object-Oriented Technologies, 1995.
142. J. J. SHILLING, J. T. STASKO. *Using animation to design, document and trace object-oriented systems.* Technical Report GIT-GVU-95-03, Georgia Institute of Technology, Graphics, Visualization and Usability Center, College of Computing, Atlanta, GA 30332-0280, 1992.

Systemy wizualizacji dla systemów równoległych

143. G. EISENHAUER, W. GU, E. KRAEMER, K. SCHWAN, J. T. STASKO. *Online Displays of Parallel Programs: Problems and Solutions.* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, USA, 1997, str. 11-20.
144. M.T. HEATH, J.A. ETHERIFGE. *Visualising the Performance of Parallel Programs.* IEEE Software, 8(5), 1991, str. 29-39.
145. E. KRAEMER, J. T. STASKO. *The Visualization of Parallel Systems: an Overview.* Journal of Parallel and Distributed Computing, Vol. 18, 1993, str. 105-117.

146. M.V. LAPOLLA. *Towards a Theory of Abstraction and Visualisation for Debugging Massively Parallel Programs.* Proceedings of the Hawaii International Conference on System Sciences, vol. 2, 1992, str. 184-195.

147. T. LEHR, Z. SEGALL, D.F. VRSALOVIC, E. CAPLAN, A.L. CHUNG, C.E. FINEMAN. *Visualising Performance Debugging.* IEEE Computer 22(10), 1989, str. 38-51.

148. D. ZERNICK, M. SNIR, D. MALKI. *Using Visualisation Tools to Understand Concurrency.* IEEE Software, 9(3), 1992, str. 87-92.

SIECI PETRIEGO

Teoria

149. C.A. PETRI. *Kommunikation mit Automaten.* Shriften des Institutes für Instrumentelle Mathematic, Bonn 1962.

Podręczniki

150. G.W. BRAMS. *Résaux de Petri, Théorie et Pratique.* Vol. 1 et 2. Editions Masson, Paris, 1982.

151. J.L. PETERSON. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., USA, 1981.

152. W. REISIG. *Petri Nets. An Introduction.* Springer Verlag, Berlin, Heidelberg, 1985. Tłumaczenie polskie: *Sieci Petriego. Wprowadzenie.* Wydawnictwa Naukowo-Techniczne, Warszawa 1988.

153. P.H. STARKE. *Petri-Netze – Grundlagen, Anwendungen, Theorie.* VEB Deutscher Verlag der Wissenschaften, Berlin 1980. Tłumaczenie polskie: *Sieci Petri – Podstawy, Zastosowania, Teoria.* Państwowe Wydawnictwo Naukowe, Warszawa 1987.

Zastosowania

154. N.D. HANSEN, K.H. MADSEN. *Petri Net Models for the Evaluation of Applicative Programs Based on λ-Expressions.* W: A. Pagnoni, G. Rozenberg (eds). *Application and Theory of Petri Nets.* Informatik Fachbereicht, 66, Springer Verlag 1983.

155. J. GRABOWSKI. *On the Analysis of Switching Curcuits by Means of Petri Nets.* EIK, 1978, 14, 12, str. 611-617.

156. P. HRUSCHKA, A. KAPPATSCH, U. KASTENS. *Net Attributed Grammars.* University of Karsluhe, Institut für Informatik, Internal Report 16/90, 1980.

157. K. JENSEN. *Coloured Petri Nets and the Invariant Method.* Th. Comp. Sc. 14(1981), str. 317-336.

158. K. JENSEN, M. KYNG, O.L. MADSEN. *Delta Semantics Defined by Petri Nets.* University of Aarhus, Internal Report PB-95, 1979.

159. J. NOE. *A Petri Net Model for the CDC 6400.* Proc. ACM SIGOPS Workshop on System Performance Evaluation, New York, ACM, 1971, str. 362-378.

160. G. ROUCAIROL, R. VALK. *Reduction of Nets and Paraltel Programs.* ACGNT, Vol. I, 1979

161. G. ROUCAIROL. *Une transformation de programmes sequentiels en programmes parallèles.* LNCS 19 (1974), str. 327-349.

162. R.M. SHAPIRO. *The Application of General Net Theory – a Personal History.* ACGNT, Vol. I, 1979.

163. R. SHAPIRO, H. SAINT. *A new Approach to Optimization of Sequential Decisions.* Annual Review in Automatic Programming, 1970, 6, 5, str. 257-288.

SYSTEMY RÓWNOLEGŁE

164. R. ATKINSON, C. HEWITT. *Parallelism and Synchronization in Actor System.* ACM Symposium on Principles of Programming Languages, Los Angeles, CA, USA, 1977.
165. M. BEN-ARI. *Principles of Concurrent* Programming. Prentice Hall (UK), Hemel Hempstead, 1983. Tłumaczenie polskie: *Podstawy programowania współbieżnego*. WNT, Warszawa, 1989.
166. C.A.R. HOARE. *Quicksort.* Computer Journal 5, 1962, str. 10-15:
167. M.J. QUINN. *Parallel sorting algorithms for tightly coupled multiprocessors.* Parallel Computing Vol. 6, 1988, str. 349-357.
168. L. RASKIN. *Performance Evaluation of Multiple Processor Systems*. Rozprawa doktorska, Carnegie-Melon University, Pittsburgh, PA, USA, 1978.

AUTOMATY KOMÓRKOWE

169. T.J. BOSSOMAIER, D.G. GREEN. *Patterns in the Sand – Computers, Complexity and Life.* Allen and Unwin, 1998.
170. E.F. CODD. *Cellular Automata*. Academic Press Inc., 1968.
171. D. G. GREEN. *Cellular Automata. A Brief Tutorial.* Witryna WWW. http://life.csu.edu.au/complex/tutorials/tutorial1.html
172. F.P. MARIN. *Conway's Life Game.* Witryna WWW i aplet w języku Java. http://bloch.ciens.ucv.ve/~felix/Java/Simulation/Conway
173. R.T. WAINWRIGHT. *Lifepage*. Witryna WWW. http://home.earthlink.net/~hilery/life
174. Artykuły poświęcone grze w życie: *Scientific American*: Oct 70, Nov 70, Jan 71, Feb 71, Mar 71, Apr 71, Nov 71, Jan 72, Dec 75, Mar 84, May 85, Feb 87, Aug. 88, Aug. 89, Sep. 89, Jan. 90; *BYTE magazine*: Sep 75, Oct 75, Dec 75, Jan 76, Dec 78, Jan 79, Apr 79, Oct 80, Jul 81.
175. *LifeLine: A Quarterly Newsletter for Enthusiasts of John Conway's Game of Life*. Numery 1-11, 1971-73.

SYSTEMY STEROWANE PRZEPŁYWEM DANYCH

176. J.B. DENNIS. *Data flow supercomputers.* Computer, Vol. 13, No. 11, 1980.
177. J.R. GURD. *The Manchester Dataflow Machine.* Computer Physics Communications, No. 37, 1985.
178. K. HWANG, F.A. BRIGGS. *Computer architecture and parallel processing.* McGraw-Hill, New York 1984.
179. S. KOZIELSKI, Z. SZCZERBIŃSKI. *Komputery równoległe – architektura, elementy oprogramowania.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1993, str. 159-180.
180. S. WĘGRZYN. *Analiza możliwości i opracowanie metod realizacji algorytmów równoległych w systemach sieciowych.* Raport z projektu badawczego nr 3 P406 011 04 (grant KBN).
181. S. WĘGRZYN, S. KOZIELSKI. *Rozproszone systemy realizacji algorytmów równoległych.* Materiały konferencyjne Zaawansowane technologie informatyczne w Nauce Polskiej, Kraków, 1995.
182. P. SZMAL. *Kilka uwag o systemach sterowanych przepływem argumentów, kanonicznych postaciach algorytmów i ich chronologizacji.* Zeszyty Naukowe Politechniki Śląskiej, seria: Informatyka, z. 26, Gliwice, 1994, str. 87-104.

183. S. WĘGRZYN. *Systemy sterowane przepływem operacji i systemy sterowane przepływem argumentów.* Zeszyty Naukowe Politechniki Śląskiej, seria: Informatyka, z. 24, Gliwice, 1993.

INŻYNIERIA PROGRAMOWANIA

184. J.M.C. BASTIEN, D.L. SCAPIN. *Ergonomic Criteria for the Evaluation of Human Interfaces.* Dostępne w wersji francuskiej: *Critères Ergonomiques pour l'Évaluation d'Interfaces Utilisateurs. Version 2.1.* Institut National de Recherche en Informatique et en Automatique INRIA, Rapport Technique N° 156, Rocquencourt, Francja, 1993.

185. G. BLAIR, J.-B. STEFANI. *Open Distributed Processing and Multimedia.* Addison-Wesley-Longman 1998.

186. P. BOOTH. *An Introduction to Human-Computer Interaction.* Lawrance Erlbaum Associates, London 1989.

187. C.M. BROWN. *Human-Computer Interface Design Guidelines*, Ablex, N.J., 1988.

188. P. COAD, E. YOURDON. *Object-Oriented Design.* Prentice Hall, Englewood Cliffs, N.J., USA, 1991. Tłumaczenie polskie: *Projektowanie obiektowe.* Yourdon Press & Oficyna Wydawnicza Read Me, Warszawa, 1994, ISBN 83-85769-18-8.

189. P. COAD, E. YOURDON. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, N.J., USA, 1990. Tłumaczenie polskie: *Analiza obiektowa.* Yourdon Press & Oficyna Wydawnicza Read Me, Warszawa, 1994, ISBN 83-85769-17-X.

190. M. GRISLIN, C. KOLSKI. *Evaluation des interfaces homme-machine lors du dévelopment des systèmes interactifs.* Technique et science informatiques. Volume 15, No 3, 1996, str. 265-296.

191. NASSI, B. SCHNEIDERMAN. *Flowchart Technique for Structured Programming. ACM SIGPLAN Notices*, 8(8), 1973, str. 12-26.

192. T. MANDEL. *The Elements of User Interface Design.* John Wiley & Sons, New York, 1997, ISBN 0-471-16267-1.

193. B. SHACKEL. *The Concept of Usability.* W: J.D. BENNETT, J.D. CASE, J. SANDELIN, M. SMITHS (eds), *Visual Display Terminals: Usability Issues and Health Concerns.* Prentice Hall, New York, 1984, str. 45-88.

ALGORYTMIKA I TEORIA INFORMATYKI

194. A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN. *Data Structures and Algorithms.* Addison-Wesley, Reading, 1983. ISBN 0-201-00023-7.

195. R. SEDGEWICK. *Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.

196. S.WĘGRZYN, J.-C.GILLE, P.VIDAL. *Developmental Systems. At the Crossroads of System Theory, Computer Science, and Genetic Engineering.* Springer-Verlag, New York, NY, USA, 1990.

197. S. WĘGRZYN. *Wykłady z Podstaw Informatyki.* Krypt Politechniki Śląskiej nr 2088. Wydawnictwo Politechniki Śląskiej, Gliwice, 1997.

198. N. WIRTH. *Algorithms + Data Structures* = Programs. Prentice Hall, Englewood Cliffs, N.J., USA, 1976. Tłumaczenie polskie: *Algorytmy + struktury danych = programy.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1980, ISBN 83-201-1120-3

POZOSTAŁE POZYCJE

199. A.V. AHO, R.SETHI, J.D. ULLMAN. *Compilers: Principles, techniques and tools.* Addison-Wesley, Reading, 1986.
200. M. GAWRYŚ. *Unix: narzędzia programistyczne yacc i lex.* Wydawnictwo PLJ, Warszawa 1993.
201. W.H. HUGGINS. *Iconic Communications.* IEEE Transactions on Education, E-14, 4, 1971, str. 158-163.
202. K. KEAHEY. *A Brief Tutorial on CORBA.* OMG 1998.
203. *ActiveX SDK.* Microsoft Corp. 1995.
204. *lex* i *yacc.* Programy narzędziowy systemu *UNIX.* Dostepna jest dokumentacja on-line typu *man* a także w formie książkowej w większości implementacji systemu *UNIX.*
205. *Microsoft Developers Network MSDN.* CD-ROM. Microsoft Corp. 1998.
206. *Microsoft® Visual Basic®.* Zintegrowane środowisko programowania. Microsoft Corporation. Version 5.0, 1997.
207. *Visual Basic 5.0 Documentation. Component Tools Guide.* Microsoft Corp. 1997.
208. *Visual C++ 5.0 Documentation. Visual C++ Tutorials: Autoclick: Automation.* Microsoft Corp. 1997.
209. *Mały Słownik Języka Polskiego.* PWN, Warszawa 1993.
210. W. DOROSZEWSKI (RED.). *Słownik Języka Polskiego.* PAN, PWN, Warszawa 1967.
211. W. SZYMCZAK (RED.). *Słownik Języka Polskiego.* PAN, PWN, Warszawa 1989.
212. *The Webster's New World Dictionary® on PowerCD®.* 3rd College Edition. Simon & Schster Inc. 1994, wydanie multimedialne na CD-ROM, wersja 2.1, Zane Publishing Inc.