

**Prace Naukowe
Wydziałowego Zakładu Informatyki
Politechniki Wrocławskiej**

1

BAZY DANYCH

Redakcja naukowa

Zygmunt Mazur

**Oficyna Wydawnicza
Politechniki Wrocławskiej**



Prace Naukowe
Wydziałowego Zakładu Informatyki
Politechniki Wrocławskiej

1

BAZY DANYCH

Redakcja naukowa
Zygmunt Mazur



Oficyna Wydawnicza Politechniki Wrocławskiej
Wrocław 2000



**WYDZIAŁOWY ZAKŁAD INFORMATYKI
WYDZIAŁ INFORMATYKI I ZARZĄDZANIA
POLITECHNIKI WROCŁAWSKIEJ**

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

Recenzenci

Tadeusz MORZY

Witold REKUĆ

Opracowanie redakcyjne materiałów

Hanna MAZUR

Projekt okładki

Zygmunt MAZUR

© Copyright by Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 2000

OFICyna WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-474-5

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 297/2000.

SPIS RZECZY

Wstęp	5
-------------	---

CZĘŚĆ I

ROZWAŻANIA TEORETYCZNE

Z. Mazur, A. Wilczek, Dynamiczne więzy integralności w Systemach Zarządzania Bazami Danych	9
W. Macyna, Z. Mazur, Temporalno-obiektowe modele baz danych	19
W. Macyna, Projekt języka zapytań dla wersyjnych obiektów hierarchicznych	37
T. Pankowski, Model bazy danych obiektów częściowo etykietowanych w procesie integrowania danych	49

CZĘŚĆ II

ROZWIĄZANIA PRAKTYCZNE

A. Wilczek, Wybrane własności obiektowe relacyjnych baz danych	87
H. Mazur, K. Michalak, Borland Visual dBase 7.0 – nowoczesne środowisko do tworzenia aplikacji bazodanowych	97

CONTENTS

Introduction	5
--------------------	---

PART I

THEORETICAL CONSIDERATIONS

Z. Mazur, A. Wilczek, On dynamic integrity constraints in RDBMS	9
W. Macyna, Z. Mazur, Temporal object-oriented database models	19
W. Macyna, The concept of query language for hierarchical objects	37
T. Pankowski, Partially tagged object database model in the process of integrating data	49

PART II

PRACTICAL SOLUTIONS

A. Wilczek, Object-oriented techniques in relational databases	87
H. Mazur, K. Michalak, Borland Visual dBase 7.0	97

*Tworzenie bazy danych jest jak stworzenie wszechświata,
tyle, że bardziej skomplikowane.
W każdym razie, kiedy stworzono wszechświat,
nie było nikogo, kto mógłby narzekać.*

Michael J. Fernandez

*Nie udaję, że rozumiem wszechświat
– jest on o wiele większy ode mnie.*

Thomas Carlyle

WSTĘP

Problematyka baz danych już od wielu lat stanowi bardzo ważny dział informatyki. Powstało dotychczas wiele prac teoretycznych i rozwiązań praktycznych dotyczących gromadzenia, wyszukiwania, bezpieczeństwa danych. Jednak nadal stawiamy sobie pytania „Jak projektować dobre bazy danych”. Opracowano tysiące projektów baz danych. Zwykle jednak po pewnym czasie baza danych, którą dotychczas uważano za solidną, zaczyna sprawiać problemy.

Większość z tych problemów możemy zaklasyfikować do którejś z dwóch kategorii: problemy związane z aplikacją lub problemy związane z danymi. Problemy związane z aplikacją powstają najczęściej gdy twórcy bazy danych nie stosują się do zaleceń poprawnych metodologii projektowania aplikacji. Takie sytuacje są częste w praktyce i należy je eliminować. Najlepszym sposobem na skuteczne rozwiązanie tego problemu jest szkolenie projektantów w zakresie nowoczesnych metod projektowania aplikacji i zaawansowanych technik programowania.

Druga kategoria problemów to problemy związane z odpowiednim modelowaniem danych. Jeśli struktura bazy danych nie będzie starannie przemyślana, wówczas baza ta nie będzie spełniać oczekiwań użytkowników i nie będzie odzwierciedlać wszystkich faktów świata rzeczywistego. Zły projekt to nie zawsze konsekwencja niepoprawnego stosowania metodologii projektowania bazy danych i nie musi źle świadczyć o projektancie. Wiele metodologii projektowania baz danych jest nadal

rozwijanych, powstają nowe modele, które umożliwiają precyzyjniej modelować fakty świata rzeczywistego. Dobrze zaprojektowana baza danych umożliwia łatwą obsługę danych, pozwala na modyfikowanie struktury danych, pozwala na efektywne wyszukiwanie informacji oraz na szybką budowę aplikacji użytkowych.

Niniejszy zeszyt naukowy stanowi rezultat pracy wykonanej w większości przez pracowników i doktorantów Wydziałowego Zakładu Informatyki Politechniki Wrocławskiej.

Wyniki przeprowadzonych badań zaprezentowano w postaci artykułów naukowych, które zostały podzielone na dwie części: teoretyczną i praktyczną.

Przedstawione prace dotyczą problemów:

w aspekcie rozważań teoretycznych:

- ❖ przedstawiania metod reprezentacji, weryfikacji i utrzymania skończonego zbioru dynamicznych więzów integralności w SZBD,
- ❖ nowych temporalno-obiektowych modeli baz danych,
- ❖ rozszerzenia języka SQL o obsługę obiektów hierarchicznych oraz obsługi wersyjności tych obiektów,
- ❖ przedstawiania modelu bezschematowej bazy danych opartego na idei obiektów etykietowanych,

oraz w aspekcie rozwiązań praktycznych:

- ❖ wybranych własności języka SQL3 oraz wzorowane na tych propozycjach rozszerzenia obiektowe w SZBD Oracle8,
- ❖ zaawansowanego programowania zorientowanego obiektowo w Borland Visual dBASE 7.0.

Autorzy składają podziękowanie Recenzentom za uwagi przedstawione w recenzjach, które zostały uwzględnione w trakcie przygotowywania ostatecznych wersji artykułów.

Redaktor naukowy
Zygmunt Mazur

CZĘŚĆ I

ROZWAŻANIA
TEORETYCZNE

Zygmunt MAZUR*,
Artur WILCZEK*

DYNAMICZNE WIĘZY INTEGRALNOŚCI W SYSTEMACH ZARZĄDZANIA BAZAMI DANYCH

Dynamiczne więzy integralności to więzy, które w przeciwieństwie do statycznych więzów integralności obejmujących pojedynczy stan bazy, obejmują sekwencję stanów bazy danych. Istnieją metody specyfikacji dynamicznych więzów integralności wykorzystujące, zaczerpnięty z logiki matematycznej rachunek zdań bądź rachunek kwantyfikatorów rzędu 1-go wzbogacony o spójniki temporalne [1], [3], [5].

Problem weryfikacji i utrzymania spójności danych, w aspekcie dynamicznych więzów integralności, bez konieczności przechowywania pełnej historii stanów bazy danych, znalazł kilka rozwiązań w modelu relacyjnym baz danych [2], [3], [4], [6].

Celem pracy jest przedstawienie metod reprezentacji, weryfikacji i utrzymania skończonego zbioru dynamicznych więzów integralności w relacyjnych SZBD.

1. WPROWADZENIE

Modelowanie współczesnych, złożonych aplikacji baz danych stawia ogromne wyzwania przed analitykami, projektantami. Obok stworzenia statycznego modelu danych konieczne jest zdefiniowanie zachowania systemu w celu opisanie poprawnych operacji na przechowywanych danych.

Więzy integralności, zapewniające spójność danych, odgrywają istotną rolę w projektowaniu aplikacji baz danych. Wyróżnia się dwie kategorie więzów integralności:

- **statyczne**
- **dynamiczne (temporalne)**

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania,
Wydziałowy Zakład Informatyki, 50-370 Wrocław, Skwer Idaszewskiego 9
mazur@ci-1.ci.pwr.wroc.pl, wilczek@ci.pwr.wroc.pl

Statyczne więzy integralności dotyczą aktualnego stanu bazy danych, na przykład „każdy kierownik jest także pracodawcą”

Dynamiczne więzy integralności mogą dotyczyć obok aktualnego także przeszłych lub przyszłych stanów bazy, na przykład

„kiedy student zrezygnuje z zajęć, nie może być powtórnie przyjęty”

Więzy statyczne są stosunkowo dobrze opracowane, omawia je także standard języka SQL2. Wydaje się tylko kwestią czasu, wobec wielu metod ich efektywnego utrzymania, szerokie zastosowanie w komercyjnych SZBD. Więzy dynamiczne pozostają nadal tematem badań.

Przy specyfikacji dynamicznych więzów integralności DIC (*Dynamic Integrity Constrains*) wykorzystuje się pojęcie **stanu** bazy danych.

Założmy, że czas jest izomorficzny do zbioru liczb naturalnych. Z każdą chwilą t możemy skojarzyć stan bazy danych D_t . W ten sposób baza danych D jest widziana jako sekwencja stanów:

$$D = (D_0, D_1, D_2, D_3, \dots).$$

Każdy stan zawiera wszystkie relacje schematu bazy.

Interpretacja stanu bazy może być dwojaka i w oparciu o pojęcia *valid-time* lub *transaction-time* oznaczać stan rzeczywistości lub stan zarejestrowany. Bazy danych rejestrujące oba czasy nazywamy **temporalnymi**, zaś uwzględniające czas w jednym z tych znaczeń **historycznymi**.

Istnieje kilka metod specyfikacji DIC. Wszystkie wykorzystują pojęcie stanu bazy danych. Wyróżnić można dwie metody:

- **FOTL** (*First-Order Temporal Logic*) - DIC specyfikuje się poprzez podanie formuł klasycznej logiki pierwszego rzędu wzbogaconej o spójniki temporalne czasu przyszłego i czasu przeszłego.
- **graf przejść** – specyfikacja DIC polega na zbudowaniu grafu skierowanego, którego węzły oznaczają stany powiązanych obiektów, zaś krawędzie warunki przejścia do nowego stanu.

2. NARZĘDZIA SPECYFIKACJI

2.1. LOGIKA TEMPORALNA

Logika Temporalna Pierwszego Rzędu (*First-order Temporal Logic FOTL*), powstała poprzez dodanie do konstrukcji klasycznej logiki pierwszego rzędu spójników temporalnych:

- czasu przeszłego: ● (pre), **since**
- czasu przyszłego: ○ (next), **until**

Spełnienie formuły FOTL w danej chwili i temporalnej bazy D definiuje się następująco:

1. Atomowa formuła **A** (tj. formuła postaci $t_1 = t_2$ lub $p(t_1, \dots, t_r)$, gdzie t_i są zmiennymi, zaś p predykatem o r argumentach) jest prawdziwa w chwili i , jeżeli jest prawdziwa D_i .
2. Jeżeli główny spójnik formuły nie jest temporalny, to obowiązują wszelkie zasady klasycznej logiki pierwszego rzędu.
3. ● **A** jest prawdziwa w chwili i , jeżeli $i > 0$ i **A** jest prawdziwe w chwili $i - 1$.
4. **A since B** jest prawdziwe w chwili i , jeżeli dla jakiegoś j , $0 \leq j < i$, **B** jest prawdziwe w chwili j , i dla każdego k , $j < k \leq i$, **A** jest prawdziwe w chwili k .
5. ○ **A** jest prawdziwe w chwili i , jeżeli **A** jest prawdziwe w chwili $i + 1$.
6. **A until B** jest prawdziwe w chwili i , jeżeli dla jakiegoś j , $j > i$, **B** jest prawdziwe w chwili j i dla każdego k , $i \leq k < j$, **A** jest prawdziwe w chwili k .

Więzy integralności specyfikuje się poprzez zamknięte formuły FOTL (bez wystąpień wolnych zmiennych).

Pomocnicze spójniki temporalne definiuje się w oparciu o podstawowe:

1. $\Diamond A \equiv \text{True until } A$ – kiedyś w przyszłości (*sometime in the future*)
2. $\Box A \equiv \neg \Diamond \neg A$ – zawsze w przyszłości (*always in the future*)
3. $\blacklozenge A \equiv \text{True since } A$ – kiedyś w przeszłości (*sometime in the past*)
4. $\blacksquare A \equiv \neg \blacklozenge \neg A$ – zawsze w przeszłości (*always in the past*)

Przykłady:

1. „zlecenie może być zrealizowane tylko raz”

$$\forall_x \Box (\text{koniec}(x) \Rightarrow \bigcirc \Box \neg \text{koniec}(x))$$

2. „zlecenie powinno być realizowane w kolejności zgłaszania” (kolejka)

$$\forall_x \forall_y \Box \neg (x \neq y \wedge \text{zgloszone}(x) \wedge \bigcirc (\neg \text{koniec}(x) \text{ until } (\text{zgloszone}(y) \wedge (\neg \text{koniec}(x) \text{ until } (\text{koniec}(y) \wedge \neg \text{koniec}(x))))))$$

2.2. GRAFY PRZEJŚĆ

Grafem przejść nazywamy $T = \langle V, E, F, v, \eta, v_0 \rangle$, gdzie:

$\langle V, E \rangle$	graf skierowany o skończonym zbiorze V węzłów i zbiorze $E \subseteq V \times V$ krawędzi,
F	zbiór węzłów końcowych,
$v: V \rightarrow N$	etykiety węzłów (lub odwzorowanie węzła na formułę temporalną),
η	etykiety krawędzi (oznaczane formułami nietemporalnymi),
v_0	węzeł początkowy.

Graf musi być deterministyczny tzn. w każdym węźle etykiety różnych wychodzących krawędzi muszą się wykluczać.

Grafy przejść wykorzystują tzw. formuły bikwantyfikowane (*biquantified*). Formuły te powstają poprzez zastąpienie zmiennych zdaniowych w temporalnym rachunku zdań (*Propositional Temporal Logic*) predykatami logiki pierwszego rzędu.

W formułach tego rodzaju dopuszczalne jest użycie wyłącznie operatora temporalnego czasu przyszłego *next* oraz:

- kwantyfikatorów temporalnych: *sometime*, *always*,
- ograniczonych kwantyfikatorów temporalnych: *sometime/always ... before/until*.

Graf przejść można skonstruować dla formuł zapisanych w logice temporalnej poprzez ich transformacje w teraźniejszą, nietemporalną część do sprawdzenia w stanie początkowym i temporalną część przyszłą z operatorem *next* do weryfikacji w pozostałej (*tail*) sekwencji stanów w oparciu o następujące temporalne reguły wywodzenia (*temporal recursion rules*):

1. **always** $\varphi \Leftrightarrow \varphi \wedge \text{next always } \varphi$
2. **sometime** $\varphi \Leftrightarrow \varphi \vee \text{next sometime } \varphi$
3. **always** $\varphi \text{ until } \psi \Leftrightarrow (\psi \wedge \varphi) \vee (\varphi \wedge \text{next}(\text{always } \varphi \text{ until } \psi))$
4. **always** $\varphi \text{ before } \psi \Leftrightarrow \psi \vee (\varphi \wedge \text{next}(\text{always } \varphi \text{ before } \psi))$
5. **sometime** $\varphi \text{ until } \psi \Leftrightarrow \varphi \vee (\neg\psi \wedge \text{next}(\text{sometime } \varphi \text{ until } \psi))$
6. **sometime** $\varphi \text{ before } \psi \Leftrightarrow \neg\psi \wedge (\varphi \vee \text{next}(\text{sometime } \varphi \text{ before } \psi))$

Transformacje formuły α należy przeprowadzać do uzyskania formuł w tzw. *disjunctive normal form* tzn. sumie formuł postaci: $\psi \wedge \text{next } \varphi$, gdzie ψ jest iloczynem prostych subformuł, czyli nietemporalnych składników α występujących

jako argumenty spójników logicznych zewnątrz kwantyfikatorów, zaś ϕ jest bądź iloczynem prostych subformuł, bądź ograniczonych operatorem temporalnym.

Przykład 1.

Dla danej formuły wejściowej

$$\phi = \text{always } (A \Rightarrow \text{sometime } B)$$

przy wykorzystaniu rachunku zdań i reguł przedstawionych wyżej, mamy:

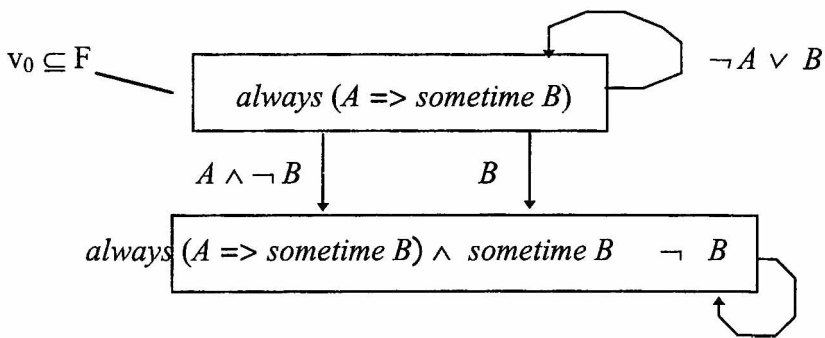
Reguła 1.

$$(\neg A \vee \text{sometime } B) \wedge \text{next } (\text{always } (A \Rightarrow \text{sometime } B))$$

Reguła 2.

$$\begin{aligned} & (\neg A \wedge \text{next } (\text{always } (A \Rightarrow \text{sometime } B))) \\ \vee & (B \wedge \text{next } (\text{always } (A \Rightarrow \text{sometime } B))) \\ \vee & ([\text{true} \wedge] \text{next } (\text{always } (A \Rightarrow \text{sometime } B)) \wedge \text{next } (\text{sometime } B)) \end{aligned}$$

Otrzymujemy graf postaci:

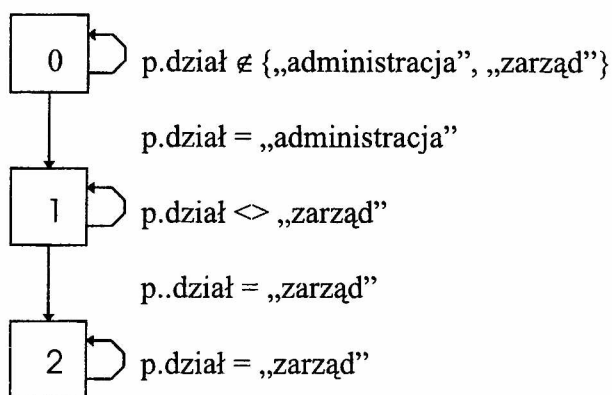


Przykład 2.

Rozważmy następujący warunek:

„zanim pracownik będzie mógł zostać członkiem zarządu, musi jakiś czas pracować w dziale administracji i kiedy zostanie wprowadzony do zarządu pozostaje w nim dopóki nie opuści firmy”.

Otrzymamy następujący graf przejść:



3. METODY WERYFIKACJI I UTRZYMANIA SPÓJNOŚCI

Metody weryfikacji DIC w klasycznych (nietemporalnych) bazach danych opierają się na pojęciu tzw. rozszerzonego stanu bazy danych D_1' .

Stan rozszerzony uzyskuje się poprzez wprowadzenie pomocniczych relacji utrzymywanych przez procesy weryfikacji DIC, co eliminuje konieczność przechowywania pełnej historii stanów bazy danych.

Relacje pomocnicze kodują historię stanów bazy danych zgodnie z zadaniem zbioru DIC. Stąd stan rozszerzony D_1' koduje całą historię ($D_0, D_1, D_2, D_3, \dots$) stanów bazy.

W danej chwili tylko jeden stan rozszerzony musi być przechowywany.

Kiedy pojawia się modyfikacja prowadząca do nowego stanu D_{i+1} , konstruowany jest nowy stan rozszerzony D'_{i+1} na bazie stanu D_{i+1} i aktualnego stanu rozszerzonego.

Więzy są weryfikowane w nowym stanie rozszerzonym. Jeżeli weryfikują się, nowy stan rozszerzony zostaje aktualnym stanem rozszerzonym, w przeciwnym wypadku aktualny stan rozszerzony nie zmienia się.

Można wyróżnić dwie metody weryfikacji DIC:

- monitor – niezależny proces weryfikujący więzy integralności
- reguły ECA – weryfikacja integralności w związku z wystąpieniem zdarzeń.

3.1. GRAFY PRZEJŚĆ

Grafy przejść są podstawowym mechanizmem służącym konstruowaniu tzw. monitorów integralności, tzn. niezależnych, od operacji na danych, procesów odpowiadających za spójność danych. Idea konstruowania tego rodzaju mechanizmów została ostatnio porzucona na korzyść metod wykorzystujących aktywne reguły (reguły *Event Condition Action*) zwane triggerami w niektórych relacyjnych SZBD. Proces utrzymania więzów integralności w oparciu o grafy przejść korzysta z pomocniczej relacji:

SITUATION ϕ (primary keys, node)

dla każdego więzu integralności ϕ , utrzymującej informacje o aktualnym znaczniku węzła grafu dla krotek relacji objętych utrzymaniem spójności względem ϕ . Dotyczy więc wszystkich relacji, których kolumny występują jako zmienne w specyfikacji ϕ . Klucze główne związanych relacji stają się kluczami zewnętrznymi relacji pomocniczej.

Przyjmijmy, że w schemacie RBD mamy relacje:

OFERTA(nr_oferty, nr_czesci, cena),

ZAMOW(nr_zamow, nr_oferty, nr_czesci, data)

oraz DIC:

„odkąd istnieje zamówienie na ofertę, jej cena nie może się zmienić, dopóki zamówienie jest realizowane oraz zamówienie musi zostać kiedyś zrealizowane”.

Wówczas relacja pomocnicza ma postać:

SITUATION(nr_zamow, nr_oferty, nr_czesci, node)

3.2. REGUŁY ECA

Istnieją metody utrzymania dynamicznych więzów integralności wykorzystujących aktywne reguły (reguły *Event Condition Action*) zwane triggerami w niektórych relacyjnych SZBD.

Chomicki [3] zaproponował specyfikację DCI za pomocą tzw. konserwatywnych formuł, czyli formuł postaci

□ ϕ , gdzie ϕ jest formuła czasu przeszłego

Dla każdej formuły ϕ stan rozszerzony bazy zawiera, dla każdej jej subformuły α (czyli dla każdej subformuły postaci $\bullet A$ lub A since B) relacje pomocnicze r_α . Dla każdej wolnej zmiennej formuły α w r_α istnieje odpowiednia kolumna.

Wiersze relacji pomocniczych utrzymywane są następująco:

α	relacja pomocnicza
$\bullet A$	$r_\alpha^0 := \text{false}$ $r_\alpha^n := A^{n-1}$
A since B	$r_\alpha^0 := \text{false}$ $r_\alpha^n := r_\alpha^{n-1} \wedge A^n \vee B^{n-1} \wedge A^n$

gdzie n oznacza odpowiedni stan bazy danych

Rozważmy przykład:

„nie można zatrudnić pracownika po jego zwolnieniu”

czyli:

$$\phi : \neg (\exists x) (\text{emp}(x) \wedge \blacklozenge (\neg \text{emp}(x) \wedge \bullet \text{emp}(x)))$$

mamy dwie subformuły:

$$\alpha_1 = \blacklozenge (\neg \text{emp}(x) \wedge \bullet \text{emp}(x))$$

$$\alpha_2 = \bullet \text{emp}(x)$$

Wiersze relacji pomocniczych utrzymywane będą następująco:

α_1	$r_{\alpha_1}^0 := \text{false}$
	$r_{\alpha_1}^n := r_{\alpha_1}^{n-1}(x) \vee (r_{\alpha_2}^{n-1}(x) \wedge \neg \text{emp}^{n-1}(x))$
α_2	$r_{\alpha_2}^0 := \text{false}$
	$r_{\alpha_2}^n := \text{emp}^{n-1}(x)$

natomiast weryfikacja formuły wyjściowej polega na sprawdzeniu wyłącznie:

$$\neg \exists x \text{ emp}^n(x) \wedge r_{\alpha 1}^n(x)$$

Reguły ECA wywoływane gdy modyfikowana jest relacja emp, powodują automatyczną weryfikację reguły j, oraz modyfikację relacji pomocniczych, bądź odrzucenie modyfikacji.

4. PODSUMOWANIE

W pracy omówiono metody specyfikacji i weryfikacji dynamicznych więzów integralności wykorzystujące logikę temporalną bądź grafy przejść. Problem weryfikacji i utrzymania spójności danych, w aspekcie dynamicznych więzów integralności, bez konieczności przechowywania pełnej historii stanów bazy danych, znalazł kilka rozwiązań w modelu relacyjnym. Przedstawione propozycje są jedną z prób zapanowania nad rosnącą złożonością modelowania zachowania we współczesnych systemach informatycznych.

LITERATURA

- [1] Manna Z., Pnueli A., *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.
- [2] Gertz M., Lipeck U.W., *Deriving Optimized Integrity Monitoring Triggers from Dynamic Integrity Constraints*, Institut für Informatik, Universität Hannover, 1996.
- [3] Chomicki J., *Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding*. TR-CS-93-16, CIS KSU, Revised, 1995.
- [4] Lipeck U.W., Zhou H., *Monitoring Dynamic Integrity Constraints on Finite State Sequences and Existence Intervals*. In J.Goers, A.Heuer, G.E.Saake (eds.), Third Workshop on Foundations of Model and Languages for Data and Objects, 1991.
- [5] Saake G., Lipeck U.W., *Using Finite-Linear Temporal Logic for Specifying Database Dynamics*. Proc. 2nd Workshop on Computer Science Logic, LNCS 385, Springer-Verlag, Berlin 1989, 288–300.
- [6] Sistla A.P., Wolfson O., *Temporal Triggers in Active Databases*. IEEE Transactions on Knowledge and Data Engineering, 1995.

ON DYNAMIC INTEGRITY CONSTRAINTS IN RDBMS

The specification of a database system consists of the description of its static information structure as well as of its dynamic behaviour. Specification of database dynamics became an important topics in the last few years. This paper presents dynamic integrity constraints, i.e. constraints on state sequences instead of single state as in the static case. In order to specify and maintain dynamic constraints we present existing approaches based on temporal logic and transition graphs.

Wojciech MACYNA*,
Zygmunt MAZUR*

TEMPORALNO-OBIEKTOWE MODELE BAZ DANYCH

W odróżnieniu od tradycyjnych baz danych gdzie informacje są modelowane w sposób statyczny, temporalna baza danych zawiera informacje zmienne w czasie. Umożliwia więc zapamiętywanie wycinków rzeczywistości nie tylko w stanie bieżącym, ale również w stanach w przeszłości. Obiektywno zorientowane modele baz danych są syntezą trzech światów: modelu złożonych wartości, semantycznych modeli bazy danych oraz koncepcji programowania obiektowego. Po badaniach nad koncepcjami reprezentowania temporalnych informacji w modelach relacyjnych przyszedł czas na badanie metod modelowania danych zmiennych w czasie zgodnie z postulatami obiektywności. W wyniku tego powstało kilkanaście modeli baz danych zwanych temporalno-obiektywnymi bazami danych.

1. WSTĘP

Artykuł składa się z kilku części. Pierwsza stanowi ogólną charakterystykę obiektowych baz danych. Jest to przedstawienie bardzo nieformalne. Formalne definicje pojęć, kierunki rozwoju oraz próby standaryzacji przedstawiono w pracach: [1], [26], [27].

Część druga opisuje czas jako zjawisko występujące w świecie rzeczywistym i wprowadza pojęcia związane z wymiarowością oraz strukturą czasu.

Część trzecia skupia się na modelach temporalno-obiektywnych, a ściślej na metodach obsługi czasu w modelach obiektowych. W opisie tym pominięto elementy wspólne, typowe i charakterystyczne dla konwencji obiektowych baz danych; szczególny nacisk położono na omówienie cech charakterystycznych dla poszczególnych modeli i jednocześnie różniących je od siebie.

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania,
Wydziałowy Zakład Informatyki, 50-370 Wrocław, Skwer Idaszewskiego 9
mazur@ci-1.ci.pwr.wroc.pl, macyna@ci.pwr.wroc.pl

2. OGÓLNA CHARAKTERYSTYKA I POSTULATY OBIEKTOWYCH BAZ DANYCH

W ostatnich latach zaproponowano wiele różnych modeli obiektowych baz danych. Ich główną cechą charakterystyczną jest zorientowanie na obiekt, który jest zdefiniowany jako para (identyfikator, wartość). Obiekty komunikują się ze sobą za pomocą metod, metody operują bezpośrednio na wartościach obiektów.

W obiektowych bazach danych można rozwiązać wiele problemów korzystając z rozwiniętych już technik w modelu relacyjnym. Z drugiej strony jednak obecność nowych cech, takich jak np. identyfikatory obiektów i metody powodują powstanie nowych problemów i technik.

Jak do tej pory nie uzgodniono jeszcze wzorcowego modelu, lecz są już prowadzone prace standaryzacyjne [28]. Opracowano jednak podstawowe postulaty [1], które każdy z tych modeli spełnia. Oto one:

1. Struktura obiektu. Obiekt składa się z ograniczonego zbioru atrybutów.
2. Identyfikacja obiektu. Każdy obiekt jest identyfikowany za pomocą identyfikatora obiektu, który jest niezależny od wartości atrybutu.
3. Obiekty złożone. Zdolność konstruowania obiektów za pomocą konstruktorów dla podstawowych typów. Minimalny zbiór konstruktorów powinien zawierać typ zbioru, typ listy oraz typ złożenia. Wartościami atrybutów obiektów mogą być inne obiekty. Użytkownik korzysta z wbudowanego mechanizmu zarządzania tymi atrybutami.
4. Metody. Obiekty mogą komunikować się ze sobą za pomocą metod. Obiekt może oddziaływać na inne obiekty tylko za pomocą metod.
5. Przeciążenie metod. Metody mogą być użyte dla różnych typów obiektów. Implementacja metod zależy od typu obiektu.
6. Typy i klasy. Pojęcie pierwsze jest używane w językach programowania do testowania poprawności programów, drugie pojawia się tylko podczas wywołania programu. Funkcjonalnie obie koncepcje są do siebie podobne.
7. Hierarchia i dziedziczenie. Wydaje się naturalne grupowanie obiektów w klasy. Dziedziczenie pozwala na konstruowanie nowej klasy na bazie już istniejącej.
8. Enkapsulacja. Użytkownik może operować na obiekcie tylko za pomocą metod a nie bezpośrednio.

3. ZJAWISKO CZASU

Czas jest bardzo istotnym zjawiskiem zarówno w świecie rzeczywistym, jak również w świecie baz danych. Zdolność do modelowania temporalnego wymiaru jest podstawą do tworzenia wielu aplikacji komputerowych bankowych, księgowych, medycznych i innych. Badania nad ujęciem informacji czasowej rozwijają się w następujących dziedzinach informatyki:

1. Systemy baz danych – spowodowane potrzebą gromadzenia, przechowywania i aktualizacji informacji czasowej.
2. Sztuczna inteligencja – na podstawie danych czasowych realizuje się rozwiązywanie zadań dotyczących planowania, diagnozowania oraz tworzenia scenariuszy.
3. Inżynieria oprogramowania – problem czasu w zakresie modelowania i formalnego opisu systemów informatycznych ujmujących dane w ich czasowym kontekście.

Narzędziami do opisywania czasu jak również do opisu zjawisk zachodzących w różnych punktach czasu są logiki temporalne. W pracy [11] przedstawiono kilkanaście takich narzędzi stworzonych przede wszystkim na potrzeby sztucznej inteligencji.

3.1 STRUKTURA CZASU

Badania nad logiką temporalną skupiają się na dwóch strukturach czasu: *liniowym* i *rozgałęzionym*. W modelu *liniowym* czas jest przedstawiony liniowo tzn. zdarzenia są uporządkowane od przeszłości do teraźniejszości w sposób rosnący. Natomiast w modelu *rozgałęzionym* – czas ma postać liniową od przeszłości do teraźniejszości, gdzie następnie może rozdzielać się na kilka linii czasowych, z których każda reprezentuje potencjalne wersje zdarzeń. Każda taka ścieżka może się rozdzielać na następne gałęzie. Tak więc struktura czasu rozgałęzionego jest drzewem, której korzeniem jest stan obecny.

Najbardziej ogólne modele czasowe w logice temporalnej przedstawiają czas jako zbiór o częściowym porządku. Dodatkowe aksjomaty mogą być wprowadzone dla bardziej wyrafinowanych modeli. Można wprowadzić całkowity porządek w zbiorze, a także cykliczność dla procesów powtarzających się w czasie.

Aksjomaty mogą być także dodane do logiki temporalnej w celu charakteryzowania gęstości linii czasowej.

Ze względu na gęstość czasową możemy modele temporalne podzielić na dwa rodzaje: *model liniowy dyskretny* i *model liniowy ciągły*. *Model liniowy dyskretny* charakteryzuje się tym, że gęstość czasu jest analogiczna do gęstości liczb naturalnych, a więc każdy punkt ma swojego następcę. Dla odróżnienia w modelu

liniowy ciągłym czas jest izomorficzny do zbioru liczb rzeczywistych tzn. pomiędzy każdymi dwoma momentami czasowymi istnieje inny moment. Najkrótszy przedział czasowy, który może być reprezentowany w danym modelu zwany jest **chronmem**.

Chociaż czas sam w sobie jest postrzegany jako coś ciągłego, trwającego, to jednak dodatkowy temporalny wymiar baz danych opiera się najczęściej na modelu liniowym dyskretnym.

Kolejnym podziałem czasu w powyższym sensie jest podział na czas relatywny i absolutny. Czasem absolutnym może być na przykład: 9.00 01.01.1993 r., podczas gdy przykładem czasu relatywnego jest przedział dziewięciu godzin. Widać przy tym, że czas absolutny ma swoje odniesienie do innego czasu. W tym przypadku jest to: północ, 01.01.0001 r. Czas relatywny różni się od odległości czasowej tym, że ma kierunek, podczas gdy odległość czasowa jest w tym sensie nieokreślona.

3.2 WYMIAROWOŚĆ CZASU

W kontekście baz danych interesujące są dwa wymiary czasowe: czas rzeczywisty (*real time*) oraz czas transakcyjny (*transaction time*). W artykule [5] zaproponowano trzeci rodzaj czasu mianowicie czas zdarzenia (*event time*).

Czas rzeczywisty – czas, w którym zdarzenie miało miejsce w świecie rzeczywistym niezależnie od czasu jego zapisania. Czas rzeczywisty może odnosić się do przyszłości jeżeli przypuszcza się, że fakty będą prawdziwe w wyspecyfikowanym czasie w przyszłości.

Czas transakcyjny – czas, w którym fakt został zapisany do bazy danych. *Czas transakcyjny* zdarzenia zaczyna się wraz z zapisaniem faktu do bazy, a kończy go transakcja, która usuwa fakt z bazy.

Czas zdarzenia – czas, w którym zaszło zdarzenie niezależnie od tego, kiedy fakt został zapisany do bazy oraz kiedy *zjawisko związane* z tym czasem było ważne w rzeczywistości. Czas ten nie jest jeszcze obsługiwany w żadnym z modeli.

W kontekście wymiarowości czasu wyróżniamy następujące modele temporalne:

Model czasu rzeczywistego – model, który obsługuje tylko czas rzeczywisty.

Model czasu transakcyjnego – model, który obsługuje tylko czas transakcyjny.

Model bitemporalny – model obsługujący oba wymiary czasu.

Przykładem faktu bazy bitemporalnej może być informacja, że osoba została przypisana do projektu na okres od czerwca do sierpnia, i że ta informacja została zapisana do bazy danych w lipcu.

3.3 RÓŻNICA POMIĘDZY CZASEM RZECZYWISTYM A TRANSAKCYJNYM

Czas transakcyjny rozpoczyna się wraz z utworzeniem bazy danych. Jego końcem jest chwila obecna, gdyż czas transakcyjny nie rozszerza się poza teraźniejszość, a fakty dotyczące przyszłości nie są zapisywane. Zmiany w bazie są robione w aktualnym czasie transakcyjnym. W miarę rozwoju stanu bazy danych czas transakcyjny rośnie monotonicznie, gdyż żadne informacje nie są nadpisywane. Czas transakcyjny zapisuje informację, kiedy dana transakcja została zapisana do bazy.

Czas rzeczywisty opisuje czas, w którym zachodzi rzeczywiście wartość jakiegoś atrybutu niezależnie od czasu zapisania do bazy.

Czas transakcyjny może poprzedzać rzeczywisty, gdy informacje o jakimś fakcie wprowadzono zanim zaszły one rzeczywiście. Czas transakcyjny późniejszy niż rzeczywisty, wskazuje na opóźnienia we wprowadzaniu informacji do bazy. Nie może on być jednak nigdy późniejszy niż czas aktualny.

Złożenie czasu transakcyjnego i rzeczywistego, a więc stworzenie bazy bitemporalnej umożliwia korekcję błędów w taki sposób, by błędne informacje nie zostały utracone. Należy podkreślić, że w transakcyjnej bazie danych wartości raz wprowadzone nie zostają usuwane, a korekcja błędów w przypadku omyłkowo zapisanych danych nie powoduje ich nadpisywania lecz wprowadza do bazy nowe wartości.

Nasuwa tutaj się pytanie po co zachowywać błędne informacje w bazie. Otóż wiadomo, że błędne informacje powodują błędne decyzje. Jeśli więc jesteśmy świadomi popełnienia błędów decyzyjnych, szukamy przyczyn tych błędów. Nie zapisując błędów do bazy danych nie wiemy co właściwie było przyczyną naszych błędnych decyzji. Tak więc błędne zapisy są traktowane jako element historii opisu obiektów. Poniżej przedstawiamy przykład.

3.4 KOREKCJA BŁĘDÓW I UAKTUALNIANIE PRZESUNIĘTE W CZASIE

Poniższy przykład wyjaśnia, jakie praktyczne skutki pociąga za sobą złożenie czasu rzeczywistego oraz czasu transakcyjnego.

Korekcja błędów

Operacja *korekcji błędów* jest operacją, która generuje różne przeszłości.

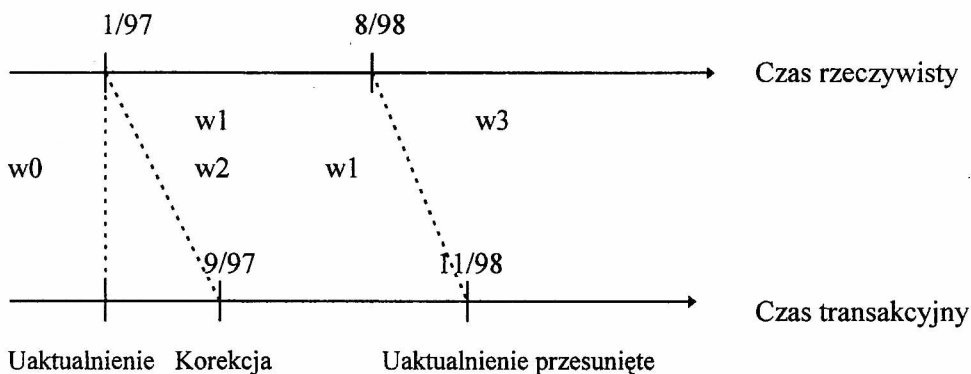
Przyjmijmy, że cena towaru t1 została podniesiona z wartości w0 do wartości w1 w styczniu 1997 r., ale została błędnie zapisana jako wartość w2 (Rys.1a). Błąd został znaleziony później i poprawiony we wrześniu 1997 r. Z korekcji błędu wynikły dwie różne przeszłości. Od stycznia 1997 r. do sierpnia 1997 r. mamy błędną przeszłość,

podczas której wartość towaru wynosi w_2 . Rysunki 1b i 1c pokazują dwie różne historie spowodowane przez tę korekcję błędu.

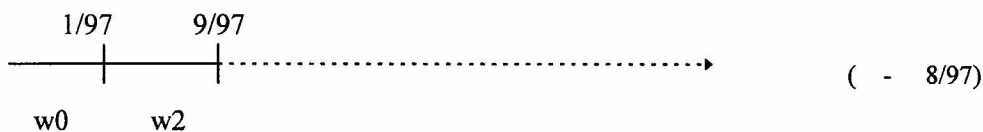
Przesunięte uaktualnianie

Przesunięte uaktualnienie w czasie zachodzi wtedy, kiedy wartość nie jest zapisana natychmiast po jej generacji. Na marginesie można powiedzieć, że każda operacja uaktualniania jest przesunięta w czasie w stosunku do zdarzenia o przedział od kilku milisekund do kilku miesięcy lub więcej. Przesunięte uaktualnienie generuje różną przeszłość w podobny sposób jak korekcja błędów. W sierpniu 1998 r. cena towaru została ustalona na wartość w_3 , ale wartość ta została zapisana dopiero w listopadzie 1998 r. W rezultacie od sierpnia 1998 r. do października 1998 r. wartość towaru może być traktowana jako w_1 lub w_3 . Rysunki 1c i 1d ukazują podwójną interpretację.

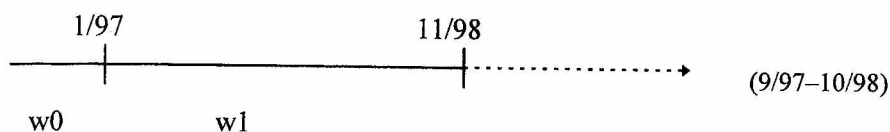
Z rysunków wynika, że używanie jedynie czasu rzeczywistego nie daje jednoznacznej wartości w punkcie czasu. Dzięki dodaniu czasu transakcyjnego możemy otrzymać wartości w dwuczasowym punkcie – pierwszą w czasie rzeczywistym, drugą w czasie transakcyjnym - bez straty przeszłości tych stanów bazy danych, które są generowane przez korekcję błędów oraz uaktualnienie przesunięte w czasie.



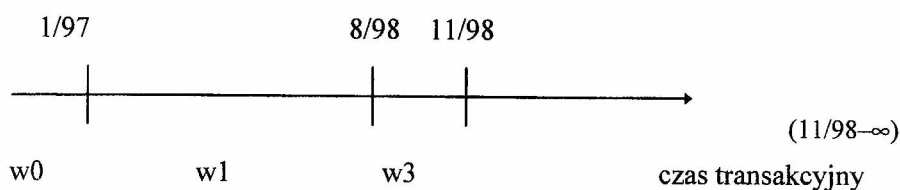
Rys. 1a. Zmiany cen towaru



Rys. 1b. Ceny do sierpnia 1997 r.



Rys. 1c. Ceny po korekcji błędu



Rys. 1d. Zmiany cen towaru

Rysunki 1b, 1c i 1d ukazują widok bazy danych z punktu widzenia czasu transakcyjnego w różnych przedziałach czasu.

3.5. CZAS ZDARZENIA

W tej części chcielibyśmy zaprezentować ciekawe podejście, wprowadzające *czas zdarzenia* (*event time*) [5]. Podejście wykazuje pewną niewystarczalność złożenia czasu rzeczywistego oraz czasu transakcyjnego w niektórych przypadkach dla pełnych informacji na temat zachodzących zjawisk w bazie danych.

Zdefiniujmy najpierw czym jest zdarzenie w kontekście tego nowego podejścia oraz jaka jest różnica pomiędzy zdarzeniem a ważnością zdarzenia.

Zdarzenie można zdefiniować jako jakiś specyficzny moment w czasie, w którym coś znaczącego się zmienia, niezależnie od tego, kiedy zajdą, bądź od kiedy już są skutki tej zmiany. Odnosząc to do baz danych zdarzenie nie jest więc jeszcze zmianą atrybutów obiektu.

Ważność zdarzenia określa kiedy pojawiają się lub od kiedy już są skutki danego zdarzenia.

Mając te dwie definicje możemy wprowadzić pojęcie **czasu zdarzenia**.

Czas zdarzenia to czas wystąpienia zdarzenia. Chociaż preferuje się, żeby w tym samym czasie, w którym zdarzenie zaistnieje, zaistniała również ważność zdarzenia to jednak istnieją sytuacje gdy tak nie jest. Opierając się na tym fakcie możemy sklasyfikować zdarzenia w trzech kategoriach: zdarzenia **natychmiastowe** (*on-time*), zdarzenia **retroaktywne** oraz zdarzenia **proaktywne**.

Mówimy o zdarzeniu *natychmiastowym* wtedy, gdy zmiana atrybutu obiektu zachodzi wraz z zaistnieniem zdarzenia.

Zdarzenie jest *retroaktywne*, jeżeli zmiana atrybutu obiektu zachodzi przed zaistnieniem zdarzenia. Przykładem może być nominacja profesorska. Przypuśćmy, że Kowalski został nominowany na profesora w sierpniu 1998 r., ale jego nominacja została ogłoszona dopiero w listopadzie 1998 r. W czasie pomiędzy sierpniem a listopadem 1998 roku Kowalski powinien mieć rangę profesora, ale w tym czasie jej nie ma, gdyż fakt nadania nominacji został ogłoszony dopiero w listopadzie. W ten sposób wartość odpowiedniego atrybutu Kowalskiego zmieniałaby się w listopadzie, lecz ważność zdarzenia zachodziłaby już wcześniej, bo od sierpnia.

W zdarzeniu *proaktywnym* (w przeciwieństwie do zdarzeń retroaktywnych) zmiana atrybutu obiektu następuje po zaistnieniu zdarzenia. Przykładem zdarzenia proaktywnego może być baza danych historii przepowiadania pogody. Podczas gdy przepowiednia pogody jest robiona, stan pogody nie istnieje jeszcze aż do czasu spełnienia prognozy.

Jeśli więc do powyższego rysunku dodalibyśmy jeszcze czas zdarzenia, otrzymalibyśmy w efekcie trzy osie czasów. W takim przypadku każdy punkt miałby trzy poziomy czasowe: czasu rzeczywistego, transakcyjnego oraz czasu zdarzenia.

3.6. CZAS UŻYTKOWNIKA

Niezależnie od trzech wyżej wymienionych czasów w literaturze można spotkać *czas użytkownika* (*user-defined time*). Charakteryzuje się tym, że semantyka wartości jest znana tylko użytkownikowi i nie jest interpretowana przez system, co go różni od *czasu rzeczywistego* i *czasu transakcyjnego*.

Czas użytkownika jest obsługiwany przez większość komercyjnych relacyjnych systemów zarządzania jako kolejny wymiar związany z atrybutami [17, 29]. Z faktu, że semantyka tego czasu znana jest jedynie użytkownikowi wynika możliwość kombinowania atrybutów czasowych (np. data, godzina) o różnej ziarnistości.

Istnieje wiele pomysłów próbujących rozwiązać problem obsługi czasu *użytkownika*. Zaproponowano utworzenie czasu jako abstrakcyjnego typu danych z jego własnym zbiorem operacji [18]. Inne podejście zarządza czasem bardziej bezpośrednio; proponuje stworzenie klas do zarządzania czasem. Podstawową korzyścią z tego płynącą jest wykorzystanie siły modelu obiektowego (podtypy,

dziedziczenie, polimorfizm z późnym wiązaniem), który może być używany do ekspresji semantyki czasu najbardziej odpowiadającej potrzebom aplikacji.

W modelu OODAPLEX [9] typ czasu jest zdefiniowany jako podtyp punktu. Dwie operacje zdefiniowane dla punktu, a więc operator równości (=) i porządku (<) mogą być dziedziczone oraz nadpisywane przez podklasy. Specyficzny wybór uporządkowanej relacji określa różne struktury czasu (np. liniową, gęstą). W tym przypadku czasowi nie jest przypisany odmienny status w modelu; jest on po prostu innym typem, który może być obsługiwany przez system lub przez aplikację.

3.7. PRZYKŁADY MODELI OBIEKTOWYCH OBSŁUGUJĄCYCH CZAS RZECZYWISTY

3.7.1. SPOSOBY REPREZENTACJI CZASU RZECZYWISTEGO

Przy rozpatrywaniu sposobów reprezentacji *czasu rzeczywistego* nasuwają się trzy główne pytania:

- Jak *czas rzeczywisty* jest reprezentowany?
- W jaki sposób fakty są powiązane z *czasem rzeczywistym*?
- Jak są reprezentowane wartości atrybutów?

Czas rzeczywisty może być reprezentowany jako:

- pojedynczy chronom – np. ('Melanie', 1993);
- przedział czasu – np. ('Melanie', Styczeń 1993, Marzec 1994);
- zbiór przedziałów czasu – np. {'Melanie', Styczeń 1993, Marzec 1994), ('Eric', Luty 1992, Grudzień 1993)}.

Jeśli chodzi o sposób powiązania faktów z *czasem rzeczywistym* wyróżniamy następujące rodzaje powiązań:

- powiązanie czasu z wartościami indywidualnego atrybutu;
- powiązanie czasu z grupami atrybutów;
- powiązanie czasu z całymi krotkami lub obiektami;
- powiązanie czasu ze zbiorem krotek, relacji lub obiektów.

Ostatni sposób nie został wykorzystany do żadnego z modeli, gdyż doprowadziłoby to do zbyt dużej redundancji danych.

3.7.2. RÓŻNICE POMIĘDZY MODELAMI

Opiszmy teraz trzy główne podejścia dodania obsługi *czasu rzeczywistego* do obiektowego modelu, bazach danych oraz związanych z nimi języków zapytań.

System OODAPLEX

Pierwsze podejście wykorzystuje bezpośrednio cechy modelu obiektowo zorientowanego i nie wymaga zmian ani w stosunku do modelu ani do języka zapytań. System typu OODAPLEX [9], który umożliwia zarządzanie takimi typami jak zbiór, krotka, funkcja itd. jest dostateczny do modelowania temporalnych informacji.

Rozważmy przykład obiektu:

type employee is object

function name (e: employee -> n: string)

function dept (e: employee -> f: (t: time -> d: department))

function projects (e: employee -> f: (t: time -> d: {project}))

Name jest funkcją, której argumentem jest employee, a wartość typem string. Funkcja dept wywoływana dla poszczególnego employee, zwraca funkcję o argumencie time i wartości department. Natomiast funkcja projects dla poszczególnego employee podaje zbiór projektów, które wykonywał pracownik w określonym czasie. Funkcja extent zwraca wszystkie obiekty wyspecyfikowanego typu.

Dla przykładu wypiszmy wszystkie oddziały, w których pracowała Melanie, gdy była zatrudniona przy projekcie OODBMS.

for the e in extent(employee) where name(e) = 'Melanie'

for the p in extent(project) where name(p) = "OODBMS"

for each t where p in projects(e)(t)

dept(e)(t)

end

end

end

W powyższym przykładzie czas jest związany z pojedynczym atrybutem (attribute-value timestamping).

Jako alternatywę podajmy podejście [8], gdzie czas jest związany z całym obiektem (object timestamping). Zakłada ono wprowadzenie funkcji state w obiekcie employee,

która przyporządkuje czasowi obiekt *snapshot_employee*, zawierający nietemporalne funkcje *dept* i *projects*.

type employee is object

function name (e: employee -> n: string)

function state (e: employee -> f: (t: time -> d: snapshot_employee))

type snapshot_employee is object

function dept (e: employee -> department)

function projects (e: employee -> {project})

Zaletą każdego z tych podejść jest to, że różne semantyki czasu rzeczywistego mogą być definiowane przez użytkownika, oraz że specyfikacja schematu i język zapytań pozostają niezależne od dodanych klauzul czasowych. Wadą tego podejścia jest fakt, że użytkownik musi poruszać się po całej osi czasu podczas specyfikowania schematu i zapytań.

Systemy rozszerzone o aspekty temporalności – model Sciore

Drugim podejściem jest rozszerzenie ogólnej obsługi modelu danych jak również języków zapytań o możliwości obsługi informacji zmiennych w czasie. Sciore [22] postulował użycie adnotacji (adnotations) do obsługi historii. Skoncentrujemy się na razie na historii, która jest zdefiniowana za pomocą czasu rzeczywistego. Propozycja ta obsługuje adnotacje poprzez skojarzenie zmiennej ukrytej z odpowiednią zmienną konwencjonalną. Każda ukryta zmienna jest powiązana z obiektem z klasy adnotacji. Obiekt *employee* może być reprezentowany w sposób następujący:

class employee inherits object =

variables

name : string

dept : HistoryFn, department;

projects : HistoryFn, {project};

end employee

W tym przypadku zmienna *name* jest nieprzypisana; natomiast zmienne *dept* i *projects* są związane z klasą adnotacyjną *HistoryFn*, która obsługuje historię. Zwykle *adnotacje* są niewidoczne dla użytkownika. Jeżeli zmienna *dept* jest dostępna, metoda z *HistoryFn* jest wywoływana na ukrytej zmiennej związanej z *dept* określając w ten sposób poszczególny oddział (*department*). W tym przypadku użytkownik jest nieświadomy o obecności ukrytej zmiennej. Z drugiej jednak strony pytanie o historię musi wyraźnie użyć różne metody klasy *adnotacji*. Przykładowe pytanie podane powyżej w języku OODAPLEX może teraz wyglądać w sposób następujący:


```
(( employee select : [ : e | e.name = 'Melanie' ] ). ↑ dept choose:
  ( project. ↑ name timeWhen : [ : n | n = 'OODBMS' ] ) ). value
```

Symbol $\uparrow$ wybiera zmienną historyczną; metody *choose* i *timeWhen* służą odpowiednio do wyboru wartości w wyspecyfikowanym czasie oraz do wyboru czasu, przy którym odpowiednie wartości są prawdziwe. Nazwa klasy natomiast zaznacza wszystkie obiekty poszczególniej klasy.

Podejście to jest wygodne jeżeli chodzi o implementację *czasu rzeczywistego*. Optymalizacja zapytań jest dalej sprawą trudną do rozwiązania, ale manipulacja w czasie jest opisywana przez funkcje użytkownika, które z reguły są indywidualnie optymalizowane.

Inne modele

System VISION jest podejściem podobnym [4]. Tak jak w OODAPLEX i w tym modelu atrybutami są funkcje. Model jest następnym krokiem do traktowania funkcji jako obiektów. Funkcje mogą mieć funkcje powiązane z nimi. Te drugie funkcje są nazwane metafunkcjami, które są bardzo podobne do metod związanych z klasami adnotacyjnymi. opisanymi powyżej.

Jako kontrast do poprzednich podejść wielu badaczy przedstawiło specjalne konstrukcje modelu danych i języków zapytań do obsługi informacji zmiennych w czasie opartych na czasie rzeczywistym. Siedem takich propozycji zostało zidentyfikowane w literaturze od 1990 roku. Przedstawmy kilka z nich a następnie porównajmy je.

Dwie propozycje są rozszerzeniem języka zapytań dla modelu relacyjnego. Rozszerzony temporalny model relacyjny Elmasriego (TEER) jest rozszerzeniem o obsługę czasu rzeczywistego w stosunku do GORDAS [10]. Dodatkowo Wu zdefiniował odrębny język zapytań bazujący na modelu E-R, oraz na SQL, nazwany SERQL [30]. Kaefer zaproponował złożony model obiektowy oraz język zapytań wraz z konstruktorami temporalnymi do zaimplementowania na nietemporalnym złożonym obiektowym modelu danych zwany MAD [13].

Pozostałe języki zapytań zostały zdefiniowane na obiektowym modelu danych.

Model danych TOODM traktuje wartości atrybutów jako sekwencje czasowe, które są parami (wartość, element temporalny) [19, 20]. Język zapytań OQL/T do obiektowo temporalnego modelu OSAM*/T dodaje opcjonalną klauzulę WHEN jak również zbiór funkcji i operacji temporalnych [25]. Sciore rozszerzył model danych EXTRA [3]; wprowadził rozróżnienie pomiędzy atrybutami wersyjnymi i niewersyjnymi oraz rozszerzył język zapytań EXCESS o elementy obsługujące czas. W końcu Gadia [6] zaproponował obiektowo-zorientowany strukturalny język zapytań (OOSTSQL), rozwinięcie języka SQL, operujący na danych przestrzennych i temporalnych. Dla rozróżnienia tych trzech języków przedstawmy wcześniej przytoczone pytanie w każdym z nich. Jak pamiętamy, chodzi o wypisanie wszystkich

oddziałów, w których pracowała Melanie, gdy była zatrudniona przy projekcie OODBMS.

Pytanie w języku TOOSQL:

```
SELECT e dept e dept vt
WHERE e.name = 'Melanie' WHEN e.dept.vt.DURING (e.projects.vt)
AND p.name = 'OODBMS' IN e.projects
```

Takie same pytanie w TMQL wygląda następująco:

```
SELECT d WHILE (p.name = 'OODBMS')
FROM e(employee) - (d(department), p(project))
WHERE e.name = 'Melanie' AND p.name = 'OODBMS'
```

W końcu w języku OOSTSQL:

```
SELECT dept(e)
RESTRICTED TO [[ 'OODBMS' IN project - name (projects(e))]]
FROM employee e
WHERE name (e) = 'Melanie'
```

3.8. PRZYKŁADOWE MODELE OPIERAJĄCE SIĘ NA CZASIE TRANSAKCYJNYM

3.8.1. RÓŻNICE POMIĘDZY CZASEM RZECZYWISTYM A CZASEM TRANSAKCYJNYM W KONTEKŚCIE OBSŁUGI DANYCH

Czas transakcyjny w odróżnieniu od *czasu rzeczywistego* dotyczy zapisania faktu do bazy danych. Tak więc zarządzanie *czasem transakcyjnym* jest konieczne, gdy chcemy uzyskać informacje na temat stanu bazy danych w przeszłości. *Czas transakcyjny* ma odmienną semantykę od *czasu rzeczywistego*. Gdy wykorzystujemy *czas rzeczywisty*, zapisujemy do bazy danych historię obiektów. Możemy np. wtedy pytać bazę danych, z którym departamentem był związany nasz pracownik w chwili obecnej oraz w przeszłości. Błędy mogłyby być poprawione poprzez zmodyfikowanie bazy danych. Jeżeli natomiast zarządzamy *czasem transakcyjnym*, zamiast historii mamy zachowane różne wersje bazy danych. Mogłyby być zachowany np. obecny dział

dla każdego pracownika. Gdyby dział się zmienił, zostałaby zapisana nowa informacja ze wskazaniem, że stara już nie jest aktualna. W takiej bazie moglibyśmy się dalej spytać, w jakim dziale jest obecnie pracownik. Nie możemy się jednak spytać, w jakim dziale dany pracownik był w przeszłości. Aby dowiedzieć się tego faktu, musimy „cofnąć” stan bazy danych do interesującej nas daty i spytać się jaki jest obecny dział dla danego pracownika.

Kontrola wersyjności jest jedną z najważniejszych funkcji umożliwiających eksploatację zintegrowanych systemów CAD/CAM oraz systemów pracujących na złożonych dokumentach. Użytkownicy takich systemów potrzebują często generować i manipulować różnymi wersjami obiektów przed wybraniem obiektu ich satysfakcjonującego.

3.8.2. RÓŻNICE POMIĘDZY MODELAMI

Tak jak w przypadku czasu rzeczywistego istnieją trzy ogólne podejścia do obsługi czasu *transakcyjnego*. Pierwszym jest użycie modelu bezpośrednio bez zmian w modelu danych czy języku zapytań. Model OODAPLEX wykorzystuje to podejście. W tym przypadku *czas transakcyjny* jest obsługiwany przez ciągłe dołączanie nowej wartości do funkcji f jeżeli wartość atrybutu się zmieni.

W podejściu drugim ogólne rozszerzenie modelu danych i języka zapytań jest wykorzystywane do gromadzenia wartości zmiennych w czasie. Podejście Sciore’a omawiane wcześniej [22] może być użyte do opracowania przeglądania oraz alternatyw opierających się na rozgałęzionym czasie transakcyjnym. Zaproponował on klasę adnotacyjną *RevisionFn*, metody jak *asOf*., *checkin*: i *checkout*: oraz klasę adnotacyjną *AlternativeFn* dla obsługi rozgałęzionego czasu *transakcyjnego*.

W podejściu trzecim model danych i język zapytań zostały zmodyfikowane dla potrzeb obsługi czasu *transakcyjnego*. W modelu MATISSE OODBMS [2] cały graf obiektów jest kopiowany przy każdej transakcji, a fizyczna implementacja dzieli się wersjami. Kaefer w swoim modelu rozróżnia atrybuty obiektowe od atrybutów wersyjnych [12]. Te ostatnie są różne dla różnych wersji. W modelu TOODM atrybut typu TS [19] jest zbiorem trójek: obiektu ID, historii i korekcy. Historia i korekty są zbiorami trójek, zawierającymi obiekt ID wartości historycznej, wartości atrybutów oraz element temporalny. Informacje zawarte w zbiorach historii i korekcy umożliwiają rekonstrukcję historii i korekty w każdym punkcie wcześniejszym czasu *transakcyjnego*. W modelu Postgres trójki są powiązane z dwoma punktami czasu *transakcyjnego* określając w ten sposób przedział, w którym są one prezentowane w bazie danych [24]. Ong i Goh [16] pokazali, że system obsługi bazy danych zarządzający liniowym czasem *transakcyjnym*, regułami, procedurami oraz pytaniami, może zarządzać także wersyjnością zawierającą symulację rozgałęzionego czasu *transakcyjnego*.

3.8.3. MODEL DANYCH ORION

Zaimplementowaliśmy w modelu bazy danych ORION model Chou-Kim [15] uzyskał chyba największą akceptację. Model ten obsługuje wersyjność obiektów. Słowo kluczowe *versionable* jest używane do wskazywania, czy klasa jest wersyjna czy niewersyjna. Wersje są automatycznie generowane przez system począwszy od numeru 1. Wersja jest związana z nazwą, punktem czasu oraz statusem. Wyróżniamy trzy rodzaje wersji:

- wersja przejściowa - może być uaktualniana w każdym czasie;
- wersja robocza - może być dzielona pomiędzy obiekty ale nie uaktualniana;
- wersja uwalniana - nie może być ani uaktualniana ani usuwana.

Wersje mogą być promowane lub degradowane wewnątrz powyższej hierarchii.

Powodem narzucenia restrykcji na uaktualnianie *wersji roboczej* jest traktowanie jej jako stałej, gdyż jedynie w ten sposób *wersja przejściowa* może być utworzona z niej. Gdyby *wersja robocza* była bezpośrednio uaktualniana, potrzebowalibyśmy cały zestaw algorytmów uaktualniających, które upewniałyby nas, że tworzone wersje nie widzą uaktualnień w *wersji roboczej*.

Istnieją dwa sposoby łączenia obiektu z innym obiektem wersyjnym: statyczne i dynamiczne. Przy łączeniu statycznym referencja obiektu zawiera pełną nazwę obiektu, identyfikator obiektu oraz numer wersji obiektu. Przy łączeniu dynamicznym potrzebny jest jedynie identyfikator obiektu; numer wersji natomiast może pozostać niewyspecyfikowany. W tym przypadku system wybiera domyślny numer wersji. Gdy tworzona jest instancja klasy wersyjnej, tworzy się obiekt ogólny dla tej instancji wraz z pierwszą wersją tego obiektu. Obiekt ten jest w istocie strukturą danych dla instancji klasy wersyjnej.

Wersje mogą być powiązane ze sobą generując acykliczny graf związany z obiektem ogólnym. Model automatycznie zachowuje dwa rodzaje relacji: relację pomiędzy obiektem wersyjnym i wszystkimi jego wersjami oraz relację pomiędzy wersją i wersjami bezpośrednio uzyskiwanymi z tej wersji.

Obsługa czasu dla obiektów wersyjnych odbywa się dzięki technice zwanej *change-notification*. Opisana ona została w [15].

3.8.4. WERSYJNOŚĆ SCHEMATU

Schemat bazy może zmieniać się w zależności od potrzeb aplikacji. Jeśli ewolucja schematu jest obsługiwana, kilka schematów może być obecnych w różnych punktach czasu transakcyjnego życia bazy. Przypuśćmy, że dnia 1 maja 1998 roku dodajemy do klasy *Pracownik* atrybut *LiczDzieci*. Powstaje nam nowa rozszerzona o ten atrybut klasa *Pracownik*, a więc i nowy schemat bazy z tą rozszerzoną klasą. Dane zapisane przed dniem 1 maja 1998 r. nie posiadają atrybutu *LiczDzieci*. Cofając więc bazę

przed tę datę cofamy jednocześnie schemat bazy danych. Działanie to jest zgodne z semantyką czasu transakcyjnego, który właśnie dotyczy zmiany stanu bazy danych.

Wersyjność schematu była badana w kontekście relacyjnych jak również obiektowych baz danych. Modele obsługujące wersyjność schematu to: MATISSE [2], Postgres [24] oraz wspomniany powyżej ORION [15].

4. PODSUMOWANIE

Artykuł ten jest próbą porównania istniejących obiektowych modeli baz danych obsługujących czas. Dwie pierwsze części to wprowadzenie, w którym przedstawiono podstawowe pojęcia związane z czasem w kontekście jego struktury i wymiarowości oraz podstawowe nieformalne postulaty obiektowych baz danych. Opisano różnicę pomiędzy czasem transakcyjnym a czasem rzeczywistym oraz czasem użytkownika. W tym miejscu warto zauważyć ciekawą koncepcję wprowadzającą czas zdarzenia. Dalej opisane zostały cechy różniące poszczególne modele od siebie. Porównane są ze sobą modele obsługujące czas transakcyjny, czas użytkownika bądź czas rzeczywisty.

Ta dziedzina baz danych jak do tej pory ma swoje niekwestionowane osiągnięcia. Dobrze zrozumiała jest struktura czasu i wymiarowość czasu. Zaproponowano kilka temporalno-obiektowych języków zapytań oraz stworzono systemy zarządzania obiektowymi bazami danych obsługujące jakiś rodzaj czasu.

Można wyróżnić także kilka problemów, które powinny być rozwiązane. Do nich zaliczymy:

- brak standardowego modelu obiektowych baz danych;
- nieformalna specyfikacja temporalno-obiektowych języków zapytań;
- brak formalnej semantyki zarówno jeśli chodzi o temporalno-obiektowy model danych jak również o języki zapytań;
- nierozwiązane problemy związane z ewaluacją oraz optymalizacją zapytań obsługujących dane zmienne w czasie;
- brak efektywnych algorytmów obsługujących temporalne złączenia i indeksowanie
- niekompletnie rozwiązane badanie więzów integralności.

LITERATURA

- [1] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*. Addison–Wesley Reading 1995.
- [2] *ADB MATISSE Versioning*. 1993, Technical Report.
- [3] Carey, M., J., DeWitt, D., J., Vandenburg, S. L. *A Data Model and Query Language for EXODUS*. Proceedings of the ACM – SIGMOD International Conference on Management of Data, Chicago. 1988.

- [4] Caruso M., Sciore E., *The Vision Object-Oriented Database Management Systems. Advances in Database Programming Languages*. ACM Press, New York. 1990.
- [5] Chakravarthy S., Kim S.-K. *Time Concepts in Temporal Databases*. Information Systems 80, 91 - 125 (1994).
- [6] Cheng, T.S., Gadia S.K., *A Seamless Object-Oriented Model for Spatio-Temporal Databases*. Technical Report TR-92-41, Computer Science Department, Iowa State University. 1992.
- [7] Clifford J., Croker A., Tuzhilin A., *On Completeness of Historical Relational Query Languages*. ACM Trans. On Database Systems Vol. 19, No.1 March 1994.
- [8] Clifford J., Croker A., *Objects in Time*. IEEE Data Engineering, vol. 7, No. 4
- [9] Dayal U., *Queries and Views in an Object-Oriented Data Model*. Proceedings of the Second Workshop on Database Programming Languages. 1989.
- [10] Elmasri R., Wiederhold G. *A Formal High-Level Query Language for the Entity-Relationship Model*. Proceedings of the Conference on Entity - Relationship Approach in Software Engineering. 1983.
- [11] Hajnicz E. *Reprezentacja logiczna wiedzy zmieniającej się w czasie*. Akademicka Oficyna wydawnicza PLJ Warszawa 1996
- [12] Kaefer W., Schoning H., *Mapping a Version Model to a Complex-Object Data Model*. Proceedings of the International Conference on Data Engineering. Tempe, Arizona. 1992.
- [13] Kaefer W., Schoning H., *Realizing a Temporal Complex-Object Data Model*. Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, 1992.
- [14] Kania K., Gałuchowski J., *Zagadnienia reprezentacji czasu w klasycznych systemach baz danych*, „Informatyka” nr 7, 1996.
- [15] Kim W. *Wprowadzenie do obiektowych baz danych*. Wydawnictwo Naukowo Techniczne Warszawa 1996.
- [16] Ong. L. P., Goh, Jeffrey K., S. *A Unified Framework for Version Modeling Using Production Rules in a Database System*. Electronics Research Laboratory, University of California. 1990.
- [17] Oracle Computer, Inc. *ORACLE Terminal User's Guide*, 1987.
- [18] Overmayer R., Stonebraker, M. *Implementation of a Time Expert in a Database System*. ACM - SIGMOD Record, Vol. 12, No. 3, 1982.
- [19] Rose E., Segev A., *TOODM – A Temporal Object-Oriented Data Model with Temporal Constraints*. Proceedings of the Tenth International Conference on the Entity Relationship Approach, 1991.
- [20] Rose E., Segev A., *TOOSQL – A Temporal Object-Oriented Query Language*. Proceedings of the Tenth International Conference on the Entity Relationship Approach, 1993.
- [21] Rose E., Segev A., *TOOA – A Temporal Object-Oriented Algebra*. Proceedings of the European Conference on Object-Oriented Programming, 1993.
- [22] Sciore E., *Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System*. ACM Transactions on Database Systems, Vol. 16, No. 3, 1991.
- [23] Snodgrass R., Ozsoyoglu G., *Temporal and Real - Time Databases: A Survey*. IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 4, August 1995.

- [24] Stonebraker M., Rowe L., *The Implementation of POSTGRES*, IEEE Transactions on Knowledge and Data Engineering, vol. 2, No. 1. 1990.
- [25] Su S.Y., Chen H.M., *A Temporal Knowledge Representation Model OSAM*T and Its Query Language OQL/T*. Proceedings of the Conference on Very Large Databases. 1991.
- [26] Subieta K., *Obiektywność w bazach danych - koncepcje, nadzieje i fakty (1)*. „Informatyka” listopad 1997.
- [27] Subieta K., *Obiektywność w bazach danych - koncepcje, nadzieje i fakty (2)*. „Informatyka” styczeń 1998.
- [28] Subieta K., *Obiektywność w programowaniu i bazach danych*. Akademicka Oficyna wydawnicza PLJ Warszawa 1998.
- [29] Tandem Computers, Inc. *ENFORM Reference Manual*. Cupertino, Cal. 1983.
- [30] Wu G., *SERQL: An ER Query Language Supporting Temporal Data Retrieval*. Proceedings of the Tenth International Phoenix Conference on Computers and Communications. 1991.

TEMPORAL OBJECT-ORIENTED DATABASE MODELS

A traditional approach to databases enables the management of only current data. The old information which is invalid at the present time is automatically deleted. In recent years some applications require a facility for modelling temporal information. Temporal databases contain and manage the history of objects. The including of various time dimensions and the existence of a new object-oriented paradigm for databases implicated the developing of temporal data models. After many proposals of extending relational models with the time attributes most researchers decided to work on models satisfying the notion of objectivity. The article describes a theoretical background and compares different data models.

Wojciech MACYNA*

PROJEKT JĘZYKA ZAPYTAŃ DLA WERSYJNYCH OBIEKTÓW HIERARCHICZNYCH

Przeważnie systemy zarządzania bazą danych obsługują jedynie obecny stan bazy, a więc zachowują informacje tylko na temat bieżących wartości danych. Okazuje się jednak, że istnieje coraz częstsza potrzeba zapamiętywania nie tylko obecnego stanu, ale również historii obiektów. Najbardziej rozpowszechnioną techniką służącą do tego celu jest technika wersyjności obiektów. Z drugiej strony obiektywne modele baz danych wprowadzają bardziej skomplikowane struktury danych niż te w tradycyjnych relacyjnych bazach danych. Nie mamy więc do czynienia z wartościami atomowymi atrybutu ale z obiektami hierarchicznymi, czyli takimi, w których wartościami atrybutu może być inny obiekt oraz zbiór wartości atomowych lub obiektów.

1. WPROWADZENIE

Artykuł jest próbą rozszerzenia języka SQL o obsługę obiektów hierarchicznych oraz obsługę wersyjności tych obiektów. Poniższy język można wzbogacić o bardziej wyrafinowane cechy wykorzystujące siłę modelu obiektywego np. wykorzystanie metod. Po wprowadzeniu artykuł zawiera przykład klas obiektywnej bazy danych, które są wykorzystywane do konstruowania przykładowych zapytań. Przedstawione dalej zapytania można podzielić na dwie kategorie. Są to pytania o stan bazy oraz pytania o historię obiektów w wyspecyfikowanym przedziale czasu. Części ostatnie to wprowadzanie funkcji zbiorowych i agregujących.

1.1. OBIEKTY HIERARCHICZNE

Pierwsza postać normalna modelu relacyjnego zakłada, że atrybut może posiadać jedynie wartość atomową. To jednak ogranicza zastosowanie tego modelu, gdyż nie jest on przydatny do modelowania nowoczesnych systemów jak np. CAD/CAM.

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania,
Wydziałowy Zakład Informatyki, 50-370 Wrocław, Skwer Idaszewskiego 9
macyna@ci.pwr.wroc.pl

W obiekcie hierarchicznym wartością atrybutu obiektu może być inny obiekt, zbiór innych obiektów lub zbiór wartości atomowych. Tworzy się w ten sposób hierarchia obiektów powiązanych za pomocą atrybutów, w której musi być obsługiwany zarówno cały obiekt hierarchiczny jak również obiekty, z których on się składa. W dalszej części artykułu są przedstawione przykłady obiektu hierarchicznego.

1.2. SPECYFIKACJA HISTORII

W literaturze spotyka się najczęściej dwa podejścia rozwiązujące problem wersyjności, która umożliwia zapis informacji dotyczących historii obiektów. Jest to wersyjność całych krotek bądź obiektów (*tuple versioning*) lub wersyjność pojedynczych atrybutów (*attribute versioning*). Podejście drugie wydaje się być lepsze, gdyż biorąc pod uwagę fakt, że atrybuty zmieniają się z różną częstotliwością, nie ma zbyt wielkiej redundancji danych. Co więcej niektóre atrybuty mogą posiadać wartość stałą, a więc powielanie ich przy zmianie dowolnego atrybutu obiektu jest bezcelowe. Poniżej przedstawiony język wykorzystuje to właśnie podejście polegające na wersyjności atrybutów obiektów.

2. PRZYKŁAD BAZY

Nasze klasy wyglądają w sposób następujący:

```
// pracownik (klasa wersyjna)
class PRACOW VERSION
{
    PNAZWA : char;           // nazwa pracownika
    PŁACA   : integer;       VERSION // płaca pracownika
    R_URODZ: integer;        // wiek pracownika
}

// projekt (klasa wersyjna)
class PROJ VERSION
{
    PRNAZWA: char;           // nazwa projektu
    BUDGT: integer;          VERSION // budżet projektu
    PRACOW: set (PRACOW); VERSION // zbiór pracowników pracujących
                                // przy projekcie
}
```

```

// wyposażenie (klasa niewersyjna)
class WYPOSAŻ
{
    LICZ_SPRZ: integer;           // liczba sprzętu
    WART : real;                  // wartość wyposażenia
}

// oddział (klasa wersyjna)
class ODDZIAŁ VERSION
{
    ONAZWA: char;                 // nazwa oddziału
    OADR: char;                   // adres oddziału
    PROJ : set ( PROJ);   VERSION // zbiór tworzonych projektów
    WYPOSAŻ : WYPOSAŻ;           // wyposażenie oddziału
}

```

Dla lepszego zobrazowania sytuacji oraz uwidocznienia hierarchicznego powiązania obiektów przedstawmy obraz naszej bazy w sposób następujący:

```

ODDZIAŁ(OID: ident, VERSION;
    ONAZWA: char;
    OADR: char;
    PROJ : (PRID: ident, VERSION
        PRNAZWA: char;
        BUDGT: integer, VERSION
        PRACOW: (PID: ident, VERSION
            PNAZWA: char;
            PŁACA: integer, VERSION
            R_URODZ : integer))
    WYPOSAŻ: (WYPID: ident
        LICZ_SPRZ: integer;
        WART: real)
)

```

gdzie OID, PRID, PID oraz WYPID oznaczają atrybuty identyfikatorów odpowiednio oddziałów, projektów, pracowników i wyposażenia.

Oczywiście bazę tę można rozszerzyć także o inne klasy w zależności od potrzeb informacyjnych użytkowników bazy. Przyjmijmy jednak, że te cztery klasy wystarczają do ukazania problematyki tego zadania oraz do jego rozwiązania.

3. JĘZYK ZAPYTAŃ

3.1. PODSTAWOWE INSTRUKCJE I ZAPYTANIA

Istnieją dwa sposoby tworzenia języka zapytań dla temporalnych baz danych. Pierwszym sposobem jest tworzenie zupełnie nowego języka od zera; drugim jest rozwinięcie już istniejącego. W tym przypadku zastosujemy drugą metodę i rozszerzymy język SQL o cechy umożliwiające tworzenie pytań o czas a także obsługę hierarchicznych obiektów.

Weźmy pod uwagę bazę danych, w której są składowane między innymi wersje oddziału „Dział programowania”. Jeżeli zmieniamy atrybut (np. zmiana adresu lub tworzenie nowego projektu) uzyskujemy nowy stan obiektu. Historia oddziału „Dział programowania” może być traktowana jako uporządkowane sekwencje stanów obiektów w czasie. Przypuśćmy, że użytkownik chce uzyskać informacje na temat stanu oddziału „Dział programowania” w czasie 01-09-91. Może to zostać sformułowane w sposób następujący:

Pytanie 1

```
SELECT      d
FROM        d IN ODDZIAŁ
            FOR time IS 01-09-91
WHERE       d.ONAZWA = "Dział programowania"
```

Część wyrażenia FROM określa, która część bazy danych ma być brana pod uwagę. Klauzula FOR definiuje czas lub przedział czasu, w którym poszukiwany jest wynik. „d” jest tak zwaną zmienną elementową i „przebiega” po wszystkich pojawieniach się obiektu klasy ODDZIAŁ. Gdy warunek części WHERE jest spełniony, wynik wyspecyfikowany jest w klauzuli SELECT. W tym przypadku wynikiem jest cały obiekt klasy ODDZIAŁ.

W tym miejscu zrobimy dwa ogólne założenia, które pozwalają użytkownikowi zainteresowanemu jedynie obecnym stanem bazy danych nie przejmować się rozszerzeniem języka o cechy temporalne. Aby to spełnić przyjmijmy zasadę, że jeżeli klauzula FOR jest ominięta, to wyrażenie „FOR time IS NOW” jest brane pod uwagę jako wyrażenie domyślne (NOW oznacza stan teraźniejszy). Po drugie czas wyspecyfikowany w klauzuli FOR jest czasem domyślnym dla atrybutów w części SELECT i WHERE.

Odnosząc tę uwagę do powyższego pytania, a więc nie używając klauzuli FOR, jako wynik otrzymamy informację na temat obecnego stanu oddziału „Dział programowania”.

Poprzednie pytanie 1 może być zapisane równoważnie za pomocą klauzuli ASOF w SELECT i WHERE. Klauzula ASOF jest związana z atrybutem i określa czas, przy którym atrybut powinien być rozpatrzony. Na przykład, pytanie o adres oddziału „Dział programowania” w czasie 01-09-91 może być sformułowane w następujący sposób:

Pytanie 2

```
SELECT      d.OADR ASOF 01-09-91
FROM        d IN ODDZIAŁ
WHERE       d.ONAZWA = "Dział programowania"
```

W następnej części wprowadzimy dwie ważne klasy pytań. Najpierw rozpatrzmy przypadek, w którym użytkownik zainteresowany jest obiektami pojawiającymi się w określonym stanie bazy danych. W tych pytaniach jest wyspecyfikowany tylko jeden punkt czasu. Drugi rodzaj dotyczy historii obiektu w bazie danych. Odcinek czasu, w którym ta historia jest rozpatrywana, specyfikuje klauzula

"FOR time IN < FROM, TO >"

gdzie FROM i TO są dwoma momentami czasu, przy czym FROM < TO.

3.2. PYTANIA O STAN BAZY

Pytanie o stan bazy daje w wyniku stan bazy danych w momencie opisanym przez klauzulę FOR z części FROM. Czas wyspecyfikowany w klauzuli jest czasem domyślnym także dla SELECT i WHERE. Wynik może jednak dotyczyć stanu atrybutów w innych momentach niż czas domyślny. Używa się wtedy związanej z atrybutem klauzuli ASOF. Może się zdarzyć, że jeden atrybut jest w czasie t_1 , a drugi t_2 . Jeżeli klauzula ASOF specyfikuje czas, w którym dany atrybut nie istnieje, to ten obiekt nie jest brany do wyniku.

W części WHERE wyspecyfikowane są warunki, które muszą być spełnione przez obiekty. Warunki mogą zawierać czas inny niż domyślny. I tutaj znowu możemy użyć klauzuli ASOF.

Rozpatrzmy to na przykładzie. Niech e , p , q oraz d będą zmiennymi zdefiniowanymi odpowiednio dla obiektów klas: PRACOW, PROJ, WYPOSAŻ i ODDZIAŁ.

Pytanie 3

Rozważmy stan bazy danych z 01-01-92 i podajmy informacje o zatrudnionym Kowalskim.

```

SELECT  e
FROM    e IN p.PRACOW,
        p IN d.PROJ,
        d IN ODDZIAŁ
        FOR time IS 01-01-92
WHERE   e.PNAZWA = "Kowalski"

```

Pytanie to rozważa obiekt ODDZIAŁ w chwili 01-01-92 w celu znalezienia pracownika „Kowalski” i wypisaniu informacji o tym pracowniku w tym czasie.

Pytanie 4

Interesuje nas ile zarobiła osoba o nazwisku Kowalski w dniu 01-01-92, przy założeniu, że pracuje ona do chwili obecnej?

```

SELECT  e.PŁACA ASOF 01-01-92
FROM    e IN p.PRACOW
        p IN d.PROJ
        d IN ODDZIAŁ
        FOR time IS NOW
WHERE   e.PNAZWA = "Kowalski"

```

Pytanie zawiera dwa czasy: NOW oraz 01-01-92. Klauzula FOR z części FROM wskazuje, że brane są pod uwagę obiekty obecnie istniejące. Jeżeli osoba o nazwisku Kowalski pracuje w jakimś oddziale to cofamy się do momentu 01-01-92, by znaleźć płacę tej osoby w tym czasie, co jest odpowiedzią na nasze pytanie. Jeżeli zatrudniony Kowalski dołączył do zakładu później niż 01-01-92, pytanie nie da żadnego rezultatu.

Pytanie 5

Jaki jest obecny status oddziałów, które w dniu 01-09-91 zawierały ponad 9 sztuk sprzętu. Oprócz tego oddział musi posiadać co najmniej jeden projekt o budżecie większym niż 800 w 01-01-92.

```

SELECT  d
FROM    d in ODDZIAŁ
        FOR time IS NOW
WHERE   (EXISTS q IN d.WYPOSAŻ:
        q.LICZ_SPRZ ASOF 01-09-91 >= 9)
        AND (EXISTS p IN d.PROJ: p.BUDGT ASOF 01-01-92 > 800)

```

Pytanie zawiera trzy momenty czasu oraz kwantyfikator szczegółowy EXISTS. Dla każdego oddziału sprawdzane są dwa warunki po klauzuli WHERE. Jeżeli są one spełnione zwracany jest obecny stan oddziału.

3.3. PYTANIA O HISTORIĘ OBIEKTÓW

Do tej pory brane były pod uwagę jedynie wersje obiektu. W istocie jednak transakcje odwzorowują bazę danych z jednego logicznego stanu w drugi. Tak więc gdy po prawidłowym zakończeniu transakcji zmienił się jeden lub więcej obiektów hierarchicznych, uzyskuje się nowy stan bazy. Stan pozostaje stały, dopóki następny nie zostaje wygenerowany. Każdy stan bazy jest ważny podczas pewnego przedziału czasowego, a więc baza danych jest traktowana jako czasowo uporządkowana sekwencja stanów.

Jeżeli użytkownika interesuje nie stan bazy, lecz historia obiektu, może pytać się o nią za pomocą konstrukcji FOR time IN < FROM, TO >. Wynik zapytania ukazuje stany obiektu w historii, czyli stan od momentu FROM aż do momentu TO. W rezultacie użytkownik otrzymuje sekwencje wersji obiektów w czasie.

W przedziale czasu <FROM, TO> baza uzyskuje wiele stanów. Stan jest wpisany do zbioru rozwiązań jako rezultat jedynie gdy jest różny od poprzedniego stanu będącego też rozwiązaniem.

Pytanie 6

Dany jest przedział czasu pomiędzy 01-09-91 a 01-12-92. Podać dla tego przedziału wszystkie płace tych użytkowników, którzy urodzili się w roku 1968.

```
SELECT  e.PŁACA
FROM    e IN p.PRACOW
        p IN d.PROJ
        d IN ODDZIAŁ
        FOR time IN <01-09-91, 01-12-92>
WHERE   e.R_URODZ = 1968
```

Pytanie działa w sposób następujący. Na początku brani są pod uwagę wszyscy pracownicy ze stanu bazy w momencie 01-09-91. SELECT nie zawiera klauzuli ASOF, czyli każda płaca jest brana pod uwagę (czas domyślny jest aktualnym czasem stanu). W rezultacie otrzymujemy zbiór wartości płac. Potem rozważamy następny stan bazy. Jeżeli teraz rezultat jest różny od rezultatu poprzedniego otrzymujemy kolejne rozwiązanie. Zauważmy, że w pytaniu nie ma żadnej informacji na temat momentu zmiany płacy. Przypuśćmy, że pracownik zarobił tę samą ilość pieniędzy

w całym okresie. Jego płaca pojawi się w wielu zbiorach w zależności od zmian płacy innych pracowników. Dlatego też wynik ostateczny składa się z uporządkowanych sekwencji zbiorów wartości płac.

Oprócz NOW słowem kluczowym jest także słowo BEGIN. Wskazuje moment tworzenia bazy danych. Jeżeli wzięty pod uwagę przedział czasowy byłby przed dniem 01-09-91, wyspecyfikowanie czasu byłoby następujące: <BEGIN, 01-09-91>. Podobnie, po dniu 01-09-91, należy wyspecyfikować <01-09-91, NOW>.

3.4. OPERATORY STARTOF, ENDOF I VALIDTIME

Wprowadźmy teraz dwa następne operatory STARTOF i ENDOF. Operatory te biorą jako parametr stan atrybutu, a zwracają moment, w którym ten stan się rozpoczyna (STARTOF) lub kończy (ENDOF). Tak więc wartość atrybutu w przedziale <STARTOF(atr), ENDOF(atr)> pozostaje niezmienna. Przedział ten opisany jest za pomocą operatora VALIDTIME(atr).

Aby wykorzystać te operatory, rozważmy to samo pytanie co poprzednio ale przyjmijmy, że każdej płacy będzie towarzyszył czas zmiany tej płacy.

Pytanie 7

```
SELECT  e.PŁACA, STARTOF(e.PŁACA)
FROM    e IN p.PRACOW
        p IN d.PROJ
        d IN ODDZIAŁ
        FOR time IN <01-09-91, 01-12-92>
WHERE   e.R_URODZ = 1968
```

Odpowiedzią na to pytanie będzie uporządkowana sekwencja zbiorów par <PŁACA, czas>. Taka odpowiedź wskazuje, że we wspomnianym czasie pracownik urodzony w 1968 roku zaczął otrzymywać daną płacę.

Pytanie 8

Rozpatrzmy pracowników, którzy urodzili się w 1968 roku. Odkryjmy moment zmiany płacy w przedziale 01-09-91 do 01-12-92. Podajmy nazwę użytkownika, nową wartość płacy oraz czas zmiany.

```
SELECT  e.PNAZWA e.PŁACA
FROM    e IN p.PRACOW
        p IN d.PROJ
        d IN ODDZIAŁ
        FOR time IN <01-09-91, 01-12-92>
```

```
WHERE    e.R_URODZ = 1968 AND
         time = STARTOF(e.PŁACA)
```

Część pytania WHERE zawiera sekwencję "time = STARTOF(e.PŁACA)" oznaczającą, że pracownik może być wpisany do odpowiedzi tylko wtedy, gdy ma miejsce jego zmiana płacy.

Tak więc dla każdego pracownika urodzonego w 1968 roku, każda zmiana płacy będzie przekazana użytkownikowi.

W naszym przykładzie PID jest identyfikatorem atrybutu PRACOW. Wobec tego STARTOF(e.PID) podaje moment, w którym pracownik dołączył do jednego z projektów. ENDOF(e.PID) podaje czas, w którym pracownik opuścił projekt lub wartość NOW, jeżeli ciągle w nim pracuje. Fakt opuszczenia projektu może być łatwo sprawdzony przez zastosowanie warunku ENDOF(e.PID) < NOW.

Pytanie 9

Rozważmy następujące pytanie:

Podajmy płace i nazwiska pracowników w momencie opuszczenia zakładu.

```
SELECT    e.PNAZWA e.PŁACA
FROM      e IN p.PRACOW
          p IN d.PROJ
          d IN ODDZIAŁ
          FOR time IN <BEGIN, NOW>
WHERE     ENDOF(e.PID) < NOW AND
          time = ENDOF(e.PID)
```

Weźmy pierwszy stan bazy danych (wskazany przez słowo kluczowe BEGIN). Dla każdego pracownika w tym stanie określa się, czy opuścił już zakład (warunek ENDOF(e.PID) < NOW) oraz czy zmienna „time” osiągnęła moment, w którym pracownik opuścił zakład. Jeżeli warunki są spełnione zostanie wypisane nazwisko i płaca danego pracownika.

3.5. WYKORZYSTANIE OPERATORÓW INCLUDE, INTERSECT, VALIDTIMEOF

Operatory INCLUDE, INTERSECT są operatorami działającymi na zbiorach. INCLUDE określa, czy jeden przedział czasowy jest zawarty w innym, INTERSECT czy część wspólna dwóch przedziałów jest pusta czy nie.

Przykładem wykorzystania operatora INTERSECT może być pytanie o takie osoby, które kiedyś jakiś nieokreślony czas pracowały z daną osobą. Znaczyłoby to, że część wspólna przedziałów czasowych pracy dwóch osób nie jest zbiorem pustym.

Pytanie 10

Sformułujmy następujące pytanie:

Podaj ostatni adres tych oddziałów, które istniały przez pewien czas jednocześnie z oddziałem „Dział programowania”.

```
SELECT  d1.OADR ASOF ENDOF(d1.OID)
FROM    d1 IN ODDZIAŁ
        FOR time1 IN <BEGIN, NOW>
WHERE   VALIDTIMEOF (d1.OID) INTERSECT
        (SELECT  VALIDTIMEOF (d2.OID)
         FROM    d2 IN ODDZIAŁ
          FOR    time2 IN <BEGIN, NOW>
         WHERE d2. ONAZWA = „Dział programowania”)
```

Powyższe pytanie jest pytaniem z podzapytaniem. Podzapytanie (po drugim słowie SELECT) określa przedział czasu, w którym oddział „Dział programowania” istnieje. Warunek WHERE jest spełniony jeśli czas istnienia oddziału posiada niepustą część wspólną z czasem istnienia oddziału „Dział programowania”. Wreszcie całe zapytanie zwraca ostatnie adresy oddziałów, które istniały przez pewien czas jednocześnie z oddziałem „Dział programowania”.

3.6. FUNKCJE LICZĄCE

Na koniec tej części przyjrzyjmy się użyciu funkcji liczących w pytaniach temporalnych. Ułożmy pytanie:

Pytanie 11

"Oblicz średnią płacę pracowników w dniu 01-01-92"

```
AVG ( SELECT  e.PŁACA
      FROM    e IN p.PRACOW
            p IN d.PROJ
            d IN ODDZIAŁ
      FOR time IS 01-01-92)
```

Dla każdego pracownika pracującego w zakładzie dnia 01-01-92 płaca jest określona i może być zastosowana funkcja AVG, licząca wartość średnią ze zbioru wartości.

Pytanie 12

Funkcja COUNT zwraca nam sumę elementów zbioru. Rozważmy ją w następującym pytaniu:

„Rozpatrzmy stan bazy danych w dniu 01-01-92. Podajmy dla każdego oddziału jego nazwę ONAZWA, nazwę projektu (PRNAZWA) ale tylko o budżecie większym niż 1000 oraz liczbę zatrudnionych pracowników”.

```
SELECT    d.ONAZWA,
          (SELECT    PRNAZWA
              COUNT (SELECT    e.PID
                          FROM    e IN p.PRACOW)
              FROM    p IN d.PROJ
              WHERE    p.BUDGT > 1000)
FROM      d IN ODDZIAŁ
FOR time IS 01-01-92
```

Na podstawie tego pytania widać, że nie tylko część WHERE może zawierać podzapytanie, ale i również SELECT.

Odpowiedzią na powyższe pytanie jest zbiór obiektów hierarchicznych. Dla każdego oddziału brane są pod uwagę wszystkie duże projekty. Dla takich projektów liczona jest liczba pracowników. W ten sposób wynik zawiera zbiór par <ONAZWA, PRNAZWA, NUMBER>, gdzie NUMBER jest liczbą zatrudnionych osób.

Rozpatrzmy to samo pytanie co poprzednio ale teraz w historii. Pytanie miałoby inną klauzulę FOR: "FOR time IN <BEGIN, NOW>". Klauzula ta oznaczałaby, że każda zmiana stanu relacji ODDZIAŁ jest brana pod uwagę. Każda zmiana mająca wpływ na wynik będzie spostrzeżona oraz będzie wygenerowany uaktualniony wynik.

Pytanie 13

Jako przykład podajmy zapytanie:

„Podaj dla każdego obecnie istniejącego oddziału nazwę i liczbę projektów w latach 1991–1992”.

```
SELECT    d1.ONAZWA,
          (SELECT    COUNT (SELECT    p.PRID
                          FROM    p IN d.PROJ)
              FROM    d IN ODDZIAŁ
              FOR    time2 IN <01-01-91, 31-12-92>
```

```

WHERE      d.ONAZWA = d1.ONAZWA)
FROM      d1 IN ODDZIAŁ
FOR time1 IS NOW.

```

Zapytanie bierze pod uwagę każdy obecnie istniejący oddział d1. Podaje najpierw jego nazwę, potem rozpatruje przedział lat 1991–1992. Następnie liczy i podaje liczbę projektów w dniu 01-01-91 i zapisuje każdą zmianę tej liczby aż do osiągnięcia daty 31-12-92.

4. WNIOSKI

Artykuł opisuje propozycję języka zapytań obsługującego obiekty hierarchiczne oraz wersyjność tych obiektów, która jest wygodnym sposobem zapisu danych z przeszłości. Język ten jest rozszerzeniem języka SQL – standardowego języka zapytań relacyjnych baz danych. Złożone obiekty hierarchiczne są obsługiwane w sposób bardzo jasny i prosty, a ich wersyjność jest rozwiązana poprzez dodanie jedynie kilku dodatkowych klauzul. Zaproponowany język może być używany jako uniwersalny język zapytań dla obiektowych modeli baz danych.

LITERATURA

- [1] Blanken H., Ijbema A., *Language Concepts for Versioned Hierarchical Objects* Temporal Aspects in Information Systems 1988.
- [2] Clifford J., Tansel A.U., *On an Algebra for Historical Relational Databases: Two Views*. Proc. of SIGMOD '85 1985, Austin.
- [3] Clifford J., Warren D.S., *Formal Semantics for Time in Databases*, ACM TODS, Vol. 8, No. 2, June 1983.
- [4] Clifford J., Croker A., Tuzhilin A., *On Completeness of Historical Relational Query Languages*. ACM Trans. On Database Systems Vol. 19, No. 1, March 1994.
- [5] Kim W., *Wprowadzenie do obiektowych baz danych*. Wydawnictwo Naukowo Techniczne Warszawa 1996.
- [6] Sarda N., *Extensions to SQL for Historical Databases* IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 20, June 1990.
- [7] Snodgrass R., *The temporal Query Language TQuel* Proc. Third ACM SIGMOD Symposium on Principles of Database Systems, April 1984.

THE CONCEPT OF QUERY LANGUAGE FOR HIERARCHICAL OBJECTS

Most of database systems manage only current state of database. There is a little support for time-varying information. Many modern database systems require a facility for modelling temporal information. The object-oriented data model which appeared recently can be applied to such databases. It enables to support more complex data structures and hierarchical objects where the value of a attribute may be not only of a primitive type but also a complex one or an another object. The concept of hierarchical objects can be supported by language described below.

Tadeusz PANKOWSKI*

MODEL BAZY DANYCH OBIEKTÓW CZĘŚCIOWO ETYKIETOWANYCH W PROCESIE INTEGROWANIA DANYCH

Celem pracy jest przedstawienie modelu bezschematowej bazy danych opartego na idei obiektów etykietowanych. Obiekty etykietowane, wywodzące się z modeli danych opartych na skierowanych grafach etykietowanych, stanowią podstawę tworzenia wielu eksperymentalnych systemów integracji danych [12] takich jak: Tsimmis [19, 20], Strudel [4], Information Manifold [10, 27], Disco [25], W3QL [7]. W pracy pokazujemy, że założenie pełnego etykietowania jest założeniem zbyt silnym. Wykazujemy, że wystarczy etykietowanie częściowe – w ten sposób powstał model obiektów częściowo etykietowanych (model PLO od ang. *Partially Labeled Objects*) [14]. Podajemy ścisłe podstawy teoretyczne tego modelu, na co składa się uniwersum obiektowo ścieżkowe (abstrakcyjny model bazy danych) oraz rachunek ścieżek (język PathCal, ang. *Path Calculus*) będący językiem tego uniwersum (analogicznie jak rachunek relacji jest językiem uniwersum relacyjnego). Pokazujemy zastosowanie modelu obiektów częściowo etykietowanych do tworzenia bazy danych (repozytorium obiektów) w systemie integrowania danych z różnorodnych źródeł. Podajemy algorytm transformacji różnorodnych struktur danych w zbiór obiektów częściowo etykietowanych.

1. WPROWADZENIE

Jednym z najintensywniej rozwijanych nurtów badań w informatyce w ostatnich latach są badania poświęcone integrowaniu danych pochodzących z różnorodnych źródeł: ze sformatowanych baz danych (relacyjnych i obiektowych) oraz danych semistrukturalnych (dokumentów elektronicznych w formacie HTML, XML, SGML, LaTeX itp.).

W przypadku konwencjonalnych baz danych podkreślane jest ich sformatowanie, co oznacza, że każda baza danych zawiera skończoną liczbę schematów, według których tworzone są stany bazy danych. Obecnie jednak coraz większą uwagę zwraca się na zasoby danych nie w pełni sformatowanych – na tak zwane dane

* Politechnika Poznańska, Wydział Elektryczny, Katedra Automatyki, Robotyki i Informatyki, Poznań, ul. Piotrowo 3A, pankowski@sol.put.poznan.pl.

semistrukuralne. Zwrócenie uwagi na dane semistrukuralne wynika z następujących powodów:

- obok sformatowanych baz danych istniały zawsze zbiory danych, których struktury nie posiadały cech pełnej regularności – były to na przykład zbiory dokumentów w systemach informacji bibliotecznych, systemach CAD/CAM, systemach danych naukowo-technicznych, pliki źródłowe takich edytorów jak Tex/LaTex, dokumenty w systemach biurowych – gdzie każdy dokument może mieć tyle cech specyficznych, że trudno jest utworzyć jeden wspólny schemat dla całej ich bazy;
- powstały nowe standardy gromadzenia i przetwarzania dokumentów, co szczególnie związane jest z rozwojem zasobów internetowych (przykładami danych tego rodzaju są dokumenty HTML, dokumenty SGML, XML, PDF);
- coraz większego znaczenia nabiera integrowanie danych z różnych źródeł (bazy danych tworzone według różnych modeli danych, pliki tekstowe, zasoby WWW, multimedia), co wymusza stosowanie modeli o niskim stopniu sformatowania.

Istotną zaletą sformatowanych baz danych jest to, że konieczność ustalenia zbioru schematów wymusza systematyczne postępowanie w procesie projektowania bazy danych. Ustalony w ten sposób schemat bazy danych, złożony ze zbioru schematów wszystkich dopuszczalnych w niej jednostek, może podlegać analizie ze względu na wewnętrzną niesprzeczność, minimalność, pełność, a także ze względu na optymalizację przewidywanych procesów przetwarzania. Obowiązujące obecnie metodyki projektowania baz danych zorientowane są na tworzenie tak rozumianego globalnego (konceptualnego) schematu bazy danych. Można jednak mówić o wadach takiego podejścia:

- ustalonych schematów często nie da się wypełnić w sposób kompletny – pojawiają się nieregularności takie jak konieczność stosowania wartości zerowych (o niejednolitej semantyce), danych domyślnych czy „nieobecnych”, co prowadzi do wielu anomalii szeroko dyskutowanych w literaturze [24];
- modele takie nie nadają się ani do reprezentowania danych semistrukuralnych, ani jako modele dla integrowania danych pochodzących z niejednorodnych źródeł danych.

Konieczność przezwyciężenia powyższych trudności doprowadziła do powstania idei tzw. bezschematowych (ang. *schemaless*) baz danych, tj. takich baz danych, gdzie nie ma oddzielnie definiowanych schematów dla jednostek danych. Dane pamiętane są łącznie z informacją pozwalającą na ich właściwą interpretację, na przykład w przypadku obiektów etykietowanych (ang. *tagged objects*) informację tę tworzą etykiety.

Bezschematowe bazy danych są szczególnie atrakcyjne w przypadku integrowania danych z heterogenicznych źródeł informacji, w tym danych semistrukuralnych, które z założenia nie posiadają oddzielnie pamiętanego schematu.

Układ pracy jest następujący:

W rozdziale 2. omawiamy architekturę systemu integrowania danych opartą na zastosowaniu osłon (ang. *wrappers*) i mediatorów (ang. *mediators*). Baza obiektów częściowo etykietowanych umiejscowiona jest jako jednolita reprezentacja danych stanowiąca wynik transformacji wykonywanych przez osłony. W rozdziale 3. w sposób nieformalny rozważamy typowe struktury danych tworzące heterogeniczne źródła informacji oraz ich reprezentację za pomocą zbioru obiektów częściowo etykietowanych. Formalna definicja modelu obiektów częściowo etykietowanych przedstawiona jest w rozdziale 4. Definiujemy tam uniwersum obiektowo-ścieżkowe oraz składnię i semantykę języka PathCal. System bazy danych obiektów częściowo etykietowanych określany jest w rozdziale 5. W rozdziale 6. rozważany jest problem strukturalizowania odpowiedzi na zapytania. Włączanie odpowiedzi do bazy danych wymaga rozszerzenia języka PathCal o reguły pozwalające sterować tworzeniem nowych obiektów i ich etykietowaniem. Proponowanym językiem o tych cechach jest język PathLog. W rozdziale 7. podajemy algorytmy transformacji struktur danych w zbiór obiektów częściowo etykietowanych. Rozważanymi strukturami są: (1) dowolne wartości złożone obejmujące w szczególności relacje znormalizowane oraz struktury o dowolnie zagnieżdżonych krotkach i zbiorach; (2) obiekty oparte na identyfikatorach, a więc obejmujące struktury obiektowych baz danych.

2. ARCHITEKTURA SYSTEMU INTEGROWANIA DANYCH

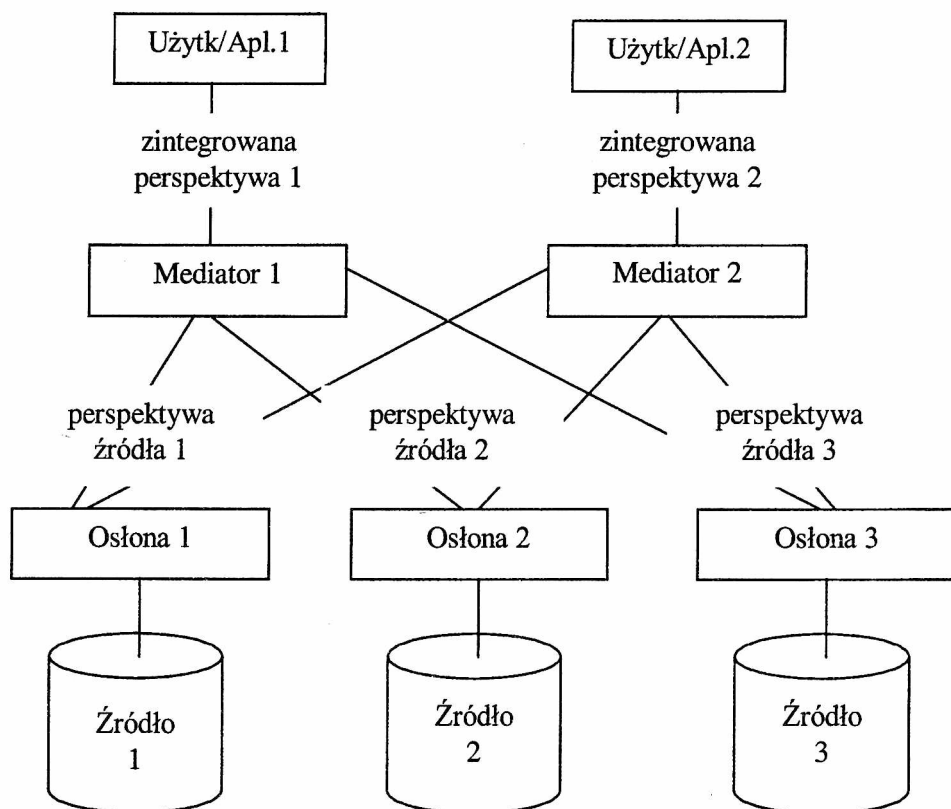
Integrowanie danych wymagane jest jako warunek wstępny realizacji takich przedsięwzięć jak na przykład:

- tworzenie problemowych repozytoriów danych i wiedzy udostępniających informacje pozyskiwane zarówno z baz danych jak i z zasobów inter- i intranetowych [27];
- budowa systemów eksploracji i analizy danych (hurtownie danych) [21];
- budowa systemów wspierających pracę zespołową, systemów wirtualnej organizacji działań i systemów kooperacyjnych [5,16,18].

W wymienionych klasach systemów *integrowania danych* podstawową rolę pełni problem ujednolicania struktury i semantyki danych pochodzących z różnorodnych źródeł. W klasycznym podejściu, gdzie integrowanie systemów autonomicznych wymaga integrowania ich schematów, wyróżniamy cztery fazy [3,12]:

1. Faza wstępna (preintegracyjna) – wykrywanie encji i związków, w tym określanie kluczy.
2. Faza porównywania – wykrywanie konfliktów nazw i konfliktów struktur.
3. Faza ujednolicania – ustalanie schematu docelowego.
4. Faza restrukturyzacji – określenie schematów i koniecznych odwzorowań.

W projektach systemów integrowania danych, w których nie zakłada się tworzenia schematu globalnego, proponowane są architektury obejmujące dwie warstwy oprogramowania – są nimi *mediatory* (ang. *mediators*) i *osłony* (ang. *wrappers*). Architekturę takiego systemu przedstawiono na rys. 1.



Rys. 1. Architektura systemu integracji danych z heterogenicznych źródeł informacji

Każdy użytkownik (lub aplikacja) może przetwarzać dane albo bezpośrednio na poziomie osłon, albo może korzystać z mediatorów dostarczających zintegrowany obraz danych eksportowanych przez osłony.

Z każdym źródłem danych związana jest osłona, której zadaniem jest udostępnienie obrazu danych związanych z tym źródłem (perspektywa źródła) zgodnego z pewnym standardem obowiązującym w całym systemie. Przekształca ona dane ze źródła w pewien wspólny model a także udostępnia wspólny język zapytań.

Z usług osłon mogą bezpośrednio korzystać użytkownicy lub aplikacje. Bardziej uniwersalnym rozwiązaniem jest jednak uwzględnienie kolejnej warstwy oprogramowania – warstwy mediatorów.

Do zadań osłon należy:

1. Konwersja źródłowych struktur danych w model poziomego osłon i na odwrót.
2. Translacja wyrażenia języka danych ze związanego z nią źródła w język danych modelu osłonowego i na odwrót. Translacja to dotyczy zarówno wyrażenia języka opisu danych, operowania danymi jak i więzów integralności bazy danych.

Mediatory odgrywają kluczową rolę w procesie integracji informacji, a ich podstawowym zadaniem jest dokonywanie *fuzji danych*. Obejmuje to grupowanie informacji (z jednego lub wielu źródeł) dotyczących jednego obiektu świata rzeczywistego, z ewentualnym usuwaniem redundancji i rozwiązywaniem niespójności między poszczególnymi źródłami danych. Mediatorom przypisuje się ponadto takie funkcje jak:

1. Selekcja właściwej informacji – optymalizacja realizacji polecenia.
2. Wspomaganie dla uzyskiwania informacji sumarycznych, wyliczalnych oraz o większym stopniu abstrakcji.
3. Wspomaganie dla wyławiania cech formalnych z niesformatowanych danych – poszukiwanie regularności i cech formalnych w danych nie w pełni sformatowanych.
4. Wspomaganie dla wykrywania niezidentyfikowanych związków w danych – zwane eksploracją danych (ang. *data mining*).
5. Wspomaganie dla udzielania odpowiedzi przybliżonych – szczególnie istotne w kooperacyjnych systemach baz danych i kooperacyjnych systemach informatycznych, gdzie można mówić o aktywnej współpracy w realizacji zespolonych zadań.

3. JEDNOLITA REPREZENTACJA DANYCH

Pierwszym etapem w procesie integrowania danych jest transformacja zawartości poszczególnych źródeł danych do jednolitego modelu. Modele tego rodzaju oparte są na idei grafów etykietowanych (na przykład model OEM/Tsimmis [22]). Do tej klasy należy również model obiektów częściowo etykietowanych (model PLO – *Partially Labeled Objects*) [14]. W podejściu tym każdy obiekt rozumiany jest jako układ:

(identyfikator, [etykieta,] wartość),

gdzie: *identyfikator* należy do zbioru OID identyfikatorów obiektów, *etykieta* jest nazwą oddającą znaczenie obiektu (jest opcjonalna), *wartość* jest albo *stałą atomową*, albo jest skończonym podzbiorem zbioru OID.

Uniwersalność tego podejścia polega na możliwości przedstawienia najważniejszych klas struktur danych – zarówno sformatowanych baz danych, jak i danych semistrukturalnych.

Następujące przykłady pokazują, jak można reprezentować wybrane struktury danych za pomocą obiektów częściowo etykietowanych (algorytmy, patrz rozdział 7).

Przykład 3.1.

Na rys. 3.1(a) przedstawiono relację znormalizowaną, a na rys. 3.1(b) – jej reprezentację za pomocą zbioru obiektów etykietowanych.

(a) R

A	B
a	b
c	d

(b)

```

1 R {3, 4}
3 A a
4 B b
2 R {5, 6}
5 A c
6 B d

```

Rys. 3.1. Relacja znormalizowana (a)
i jej przedstawienie za pomocą zbioru obiektów etykietowanych (b)

1. Rozważmy przykłady obiektów charakteryzujące się silnym typowaniem. Po lewej stronie znaku „::” zapisany jest obiekt, a po prawej – typ obiektu:

(1, [Imię : Jan, Współmałżonek: 2, Dzieci : 3]) :: (^Osoba, Osoba),

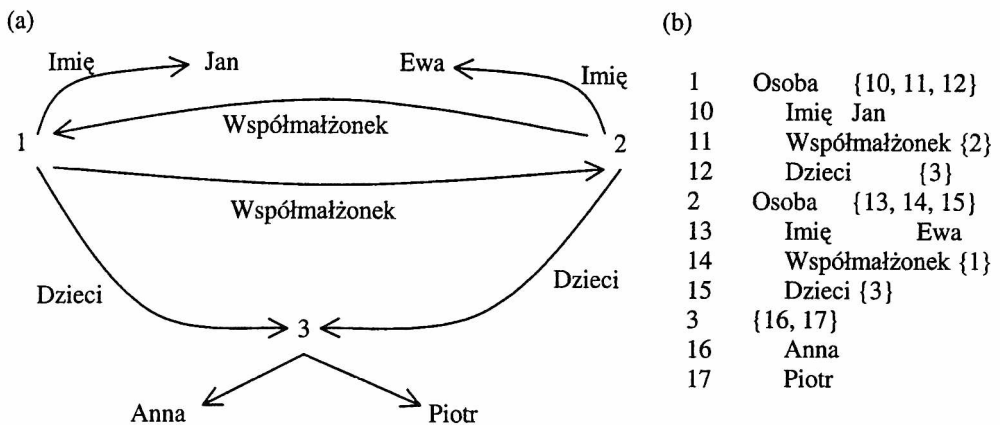
(2, [Imię : Ewa, Współmałżonek: 1, Dzieci : 3]) :: (^Osoba, Osoba),

(3, {Anna, Piotr}) :: (^{STRING}, {STRING}),

przy czym STRING jest typem atomowym, a Osoba jest typem rekurencyjnym o następującej definicji:

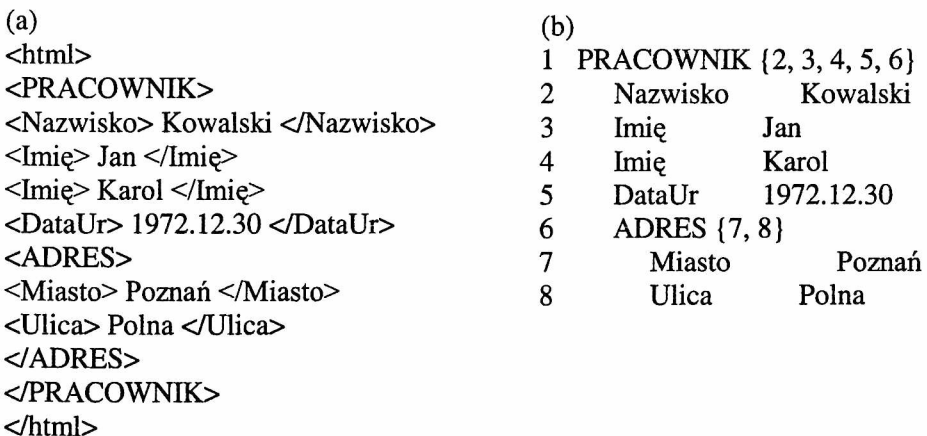
Osoba = [Imię : STRING, Współmałżonek : ^Osoba, Dzieci : ^{STRING}].

Na rys. 3.2. przedstawiono graficzną postać rozważanego zbioru obiektów oraz reprezentację tego zbioru za pomocą obiektów częściowo etykietowanych. Przy przyjętej interpretacji obiektów dzieci brak jest nazw, które można byłoby użyć jako etykiet dla obiektów o identyfikatorach 3, 16 i 17.



Rys. 3.2. Struktura obiektów (a) i jej reprezentacja za pomocą zbioru obiektów częściowo etykietowanych

Na rys. 3.3 przedstawiono przykład danych semistrukturalnych (dokument HTML) oraz jego reprezentację jako zbiór obiektów etykietowanych.



Rys. 3.3. Dokument HTML: dane semistrukturalne (a) i ich reprezentacja za pomocą zbioru obiektów etykietowanych

Struktura nieregularna z rys. 3.4(a) zapisana jest jako wartość złożona na rys. 3.4(b) oraz przedstawiona jest jako zbiór obiektów częściowo etykietowanych na

rys. 3.4(c). Zauważmy, że dla obiektu o identyfikatorze 3 nie można określić etykiety – odpowiada on tablicy krotek identyfikowanych za pomocą 4 i 5, identyfikatory 4 i 5 również nie posiadają etykiet.

(a)

Projekt:

Nazwa: A	Wykonawca: a	Godziny: 200		
	Wykonawca: b	Koszt: 1000		
Nazwa: B	Wdrażanie:	Kierownik: c	Koszt: 500	Firma: X
		Koszt: 1500	Firma: Y	

(b)

Projekt : {[Nazwa : A, {[Wykonawca : a, Godziny : 200],
[Wykonawca : b, Koszt : 1000]}],
[Nazwa : B, Wdrażanie : {[Kierownik : c, Koszt : 500, Firma : X],
[Kierownik : d, Koszt : 1500, Firma : Y]}]}

(c)

1	Projekt	{2, 3}	10	Projekt	{11, 12, 13}
2	Nazwa	A	11	Nazwa	B
3		{4, 5}	12	Wdrażanie	{14, 15, 16}
4		{6, 7}	14	Kierownik	c
6	Wykonawca	a	15	Koszt	500
7	Godziny	200	16	Firma	X
5		{8, 9}	13	Wdrażanie	{17, 18}
8	Wykonawca	b	17	Koszt	1500
9	Koszt	1000	18	Firma	Y

Rys. 3.4. Struktura nieregularna (a), jej przedstawienie jako wartości złożonej (b) i za pomocą zbioru obiektów częściowo etykietowanych (c)

Istotną cechą modelu *PLO* jest dysponowanie językiem służącym do definiowania, przeszukiwania i restrukturalizacji bazy danych obiektów częściowo etykietowanych [14,17]. Pozwala to na realizację funkcji związanych z udostępnianiem danych różnym użytkownikom w systemie kooperacyjnym, jak również na wymianę informacji między użytkownikami z automatyczną translacją postaci danych.

4. MODEL OBIEKTÓW CZĘŚCIOWO ETYKIETOWANYCH – MODEL PLO

Model PLO (ang. Partially Labeled Objects) określamy definiując dwie jego składowe, którymi są:

1. Uniwersum obiektowo-ścieżkowe złożone z obiektów, ścieżek (reprezentujących powiązania między obiektami) oraz z operacji na ścieżkach i zbiorach ścieżek.
2. Język PathCal (rachunek ścieżek) rozumiany jako język uniwersum obiektowo-ścieżkowego (oraz jego rozszerzenia).

4.1. UNIWERSUM OBIEKTOWO-ŚCIEŻKOWE

Definicja 4.1 (*obiekt*).

Niech dane będą: ID - zbiór *identyfikatorów*, D - zbiór *stałych atomowych* (złożony z liczb, ciągów znaków, dat, obrazów itp.).

Obiektem nazywamy parę $o = (i, \text{val}(i))$,

gdzie $i \in ID$ jest *identyfikatorem* obiektu, a $\text{val}(i) \in D$ lub $\text{val}(i) \subseteq ID$ jest wartością (stanem) obiektu (jeśli $\text{val}(i) \in D$, to mówimy o obiekcie *atomowym*, w przeciwnym przypadku – o obiekcie *złożonym*).

■

W dalszym ciągu jako identyfikatorów obiektów używać będziemy liczb naturalnych, a z kontekstu będzie jednoznacznie wynikać, czy dana liczba naturalna oznacza identyfikator czy wartość obiektu.

Przykład 4.1.

Obiekty atomowe:

(1, „Kowalski”), (2, 1999.07.05), (3, 2800.00),

obiekty złożone:

(4, {1, 2, 3}), (5, {4}).

■

Definicja 4.2 (*ścieżka*).

Niech OID będzie zbiorem identyfikatorów obiektów.

Ścieżką nad OID nazywamy wyróżnioną ścieżkę zerową nil (o długości 0) oraz każdy ciąg $p = (i_1, i_2, \dots)$, w którym element i_k , $k = 1, 2, \dots$, jest identyfikatorem należącym do OID , a ponadto:

wszystkie identyfikatory są różne,

$i_{k+1} \in \text{val}(i_k)$, $k = 1, 2, \dots$, tj. następnik należy do wartości poprzednika.

■

Zbiór wszystkich ścieżek nad **OID** oznaczać będziemy przez **OID*** i nazywać będziemy *domknięciem* zbioru **OID**.

Przykład 4.2.

Ścieżkami nad zbiorem identyfikatorów obiektów z przykładu 4.1. są:

- nil – wyróżniona ścieżka zerowa,
 - 1, 2, 3, 4, 5 – ścieżki o długości 1,
 - (4, 1), (4, 2), (4, 3), (5, 4) – ścieżki o długości 2,
 - (5, 4, 1), (5, 4, 2), (5, 4, 3) – ścieżki o długości 3.
- Są to wszystkie ścieżki nad zbiorem rozważanych obiektów.

Definicja 4.3 (złożenie ścieżek).

Niech p i p' będą ścieżkami nad **OID**. Złożenie ścieżek p i p' , $p \parallel p'$, definiujemy następująco (concat(p, p') jest konkatenacją ciągów p i p'):

$p \parallel p' =$ if $p = \text{nil}$ or $p' = \text{nil}$ or concat(p, p') $\notin$ **OID***
 then nil
 else concat(p, p').

Przykład 4.3.

Dla ścieżek z przykładu 4.2 otrzymujemy:

$\text{nil} \parallel 1 = \text{nil}$, $1 \parallel 1 = \text{nil}$, $(5, 4) \parallel 1 = (5, 4, 1)$, $(5, 4) \parallel (4, 1) = \text{nil}$.

Łatwo stwierdzić, że operacja złożenia ścieżek jest łączna, tj.

$$(p \parallel p') \parallel p'' = p \parallel (p' \parallel p'').$$

Operację złożenia ścieżek wykorzystujemy do zdefiniowania operacji złożenia zbiorów ścieżek.

Definicja 4.4 (złożenie zbiorów ścieżek).

Niech P i P' będą zbiorami ścieżek nad **OID**. Złożenie zbiorów ścieżek P i P' , $P \bullet P'$, definiujemy następująco:

$$P \bullet P' = \{p \parallel p' \mid p \in P, p' \in P', p \parallel p' \neq \text{nil}\}.$$

Przykład 4.4.

Dla zbiorów ścieżek z przykładu 4.2, mamy:

$$\{1, (5, 4)\} \bullet \{1, (4, 1), 2\} = \{(5, 4, 1), (5, 4, 2)\}.$$

Ważną rolę odgrywają dwie kolejne operacje na zbiorach ścieżek:

- operacje „wydłużenia ścieżek o jeden element” (oznaczana symbolem ?)
- operacje domknięcie *.

Definiujemy je następująco:

Definicja 4.5 (*operacja wydłużenia i domknięcia*).

Niech P będzie zbiorem ścieżek o skończonej długości nad **OID**. Wówczas:

$$P? = P \bullet \mathbf{OID},$$

$$P^* = P \cup (P?)^*.$$

■

Zbiór $P?$ składa się więc ze zbioru wszystkich takich ścieżek, które stanowią wydłużenie ścieżek należących do zbioru P o jeden element. Natomiast zbiór P^* zawiera wszystkie ścieżki, których dowolny przedrostek należy to zbioru P , a więc zawiera zbiór P oraz wydłużenia ścieżek z P o dowolną liczbę elementów (domknięcie P względem **OID**).

Zdefiniowane pojęcia wykorzystamy obecnie do zdefiniowania formalnej struktury matematycznej stanowiącej abstrakcyjny model bazy danych (uniwersum interpretacji języka bazy danych).

Definicja 4.6 (*uniwersum obiektowo-ścieżkowe*).

Niech **ID** będzie zbiorem identyfikatorów, **D** niech będzie zbiorem stałych atomowych. *Uniwersum obiektowo-ścieżkowym* nazywamy strukturę matematyczną

$$\mathbf{OP}_{\mathbf{ID}, \mathbf{D}} = (\mathbf{OID}, \mathbf{P}, \text{val}, \text{nil}, \parallel, \bullet, ?, *),$$

taką, że:

- $\mathbf{OID} = \text{dom}(\text{val}) \subseteq \mathbf{ID}$, zbiór identyfikatorów obiektów,
- $\mathbf{P}$ - zbiór wszystkich ścieżek nad **OID**, $\mathbf{P} = \mathbf{OID}^*$.
- $\text{val} : \mathbf{OID} \rightarrow \mathbf{D} \cup 2^{\mathbf{OID}}$, funkcja definiująca stany obiektów,
- $\text{nil} \in \mathbf{P}$ – wyróżniona ścieżka zerowa o długości 0,
- $\parallel$ – operacja złożenia ścieżek,
- $\bullet$ – operacja złożenie zbiorów ścieżek,
- $?$ – operacja wydłużenie ścieżek z zadanego zbioru o jeden element,
- $*$ – operacja domknięcia zbioru ścieżek.

■

4.2. JĘZYK PATHCAL

Język *PathCal* (ang. *Path Calculus*) traktujemy jako język uniwersum obiektowo-ścieżkowego, a więc jako język bazy danych. Dalej zdefiniujemy jego alfabet oraz zbiór dopuszczalnych wyrażeń: termy, formuły i zapytania.

Przyjmujemy, że *PathCal* jest językiem, w którym każdy term ma jeden z pięciu typów:

Typy:

1. U – typ stałych atomowych,
2. OID – typ identyfikatorów,
3. $PATH$ – typ ścieżek,
4. $\{OID\}$ – typ zbiorów identyfikatorów,
5. $\{PATH\}$ – typ zbiorów ścieżek.

Alfabet:

1. Stałe i zmienne *atomowe* (typu U), *identyfikacyjne* (typu OID) i *ścieżkowe* (typu $PATH$).
2. Symbole funkcyjne (na bytach indywidualnych):
 val – typu $OID \rightarrow D \cup \{OID\}$;
 $\cdot$ – typu $PATH \times PATH \rightarrow PATH$.
3. Symbole funkcyjne (na bytach zbiorowych):
 $\cdot$ – typu $\{PATH\} \times \{PATH\} \rightarrow \{PATH\}$;
 $?$ – typu $\{PATH\} \rightarrow \{PATH\}$;
 $*$ – typu $\{PATH\} \rightarrow \{PATH\}$.
4. *Etykiety* (predykaty 1-członowe) $L = \{L_1, \dots, L_n\}$, każda typu $\{OID\}$.
5. *Symbole relacyjne*: $=, \neq, \in, \subseteq$.
6. *Symbole logiczne*: $\neg, \wedge, \vee, \Rightarrow, \exists, \forall$.

Termy:

3. Każda stała i każda zmienna typu τ jest termem typu τ .
4. Każda etykieta L jest termem typu $\{OID\}$.
5. Jeśli $p_1, \dots, p_n$ są termami typu OID , to $\{p_1, \dots, p_n\}$ jest termem typu $\{OID\}$.
6. Każdy term typu OID jest termem typu $PATH$, a każdy term typu $\{OID\}$ jest termem typu $\{PATH\}$.
7. Jeśli p i p' są termami typu $PATH$, to $p.p'$ jest termem typu $PATH$.
8. Jeśli P i P' są termami typu $\{PATH\}$, to $P.P'$, $P?$ i $P*$ są termami typu $\{PATH\}$.
9. Każde zapytanie q jest termem typu $\{PATH\}$.

Formuły atomowe:

1. Jeśli t jest termem typu OID , a s jest termem typu U lub $\{OID\}$, to $val(t) = s$ jest formułą atomową.
2. Jeśli P jest termem typu $\{PATH\}$, a p jest termem typu $PATH$, to $P(p)$ jest formułą atomową.
3. Jeśli a jest stałą atomową, a p i p' są termami typu $PATH$, to:
 $p = a, p = p', p \neq p', p \in p'$ oraz $p \subseteq p'$ są formułami atomowymi.

Formuły:

1. Każda formuła atomowa jest formułą.
2. Jeśli φ i φ' są formułami, to formułami są także:
 $\neg\varphi, \varphi \wedge \varphi', \varphi \vee \varphi', \varphi \Rightarrow \varphi'$.
3. Jeśli φ jest formułą, a x jest zmienną typu *PATH* wolną w φ , to
 $\exists x \varphi$
 oraz
 $\forall x \varphi$
 są formułami.

Zapytania:

Jeśli p jest termem typu *PATH*, a φ jest formułą, to $\{p \mid \varphi\}$ jest zapytaniem typu $\{\text{PATH}\}$.

Następujący przykład ilustruje stosowanie podstawowych konstrukcji języka *PathCal*.

Przykład 4.5.

Niech dana będzie baza danych obiektów częściowo etykietowanych. W przykładzie tym każda trójka (i, L, v) jest *obiektem etykietowanym* i stwierdza fakt, że obiekt (i, v) przypisany jest etykietcie L występującej w alfabecie języka *PathCal*. Taki obiekt etykietowany traktujemy więc jako skrócony zapis formuły:

$$L(i) \wedge \text{val}(i) = v.$$

- | | | |
|----|-----------|----------------|
| 1 | Osoba | {2, 3, 6} |
| 2 | Nazwisko | „Nowak” |
| 3 | Adres | {4, 5} |
| 4 | Miasto | „Poznań” |
| 5 | Ulica | „Pólna” |
| 6 | Kierownik | {7} |
| 7 | Osoba | {8, 9, 10, 11} |
| 8 | Nazwisko | „Kowalski” |
| 9 | Miasto | „Poznań” |
| 10 | Ulica | „Kwiatowa” |
| 11 | Podwładny | {1} |

Rozważmy wyrażenia związane z tą bazą danych.

1. Formuły definiujące obiekty (zauważmy, że w rozważanym wystąpieniu istnieje cykl):
 $\text{Osoba}(1) \wedge \text{val}(1) = \{2, 3, 6\},$
 $\text{Nazwisko}(2) \wedge \text{val}(2) = \text{„Nowak”},$

Kierownik(6) $\wedge$ val(6) = {7},

Podwładny(11) $\wedge$ val(11) = {1},

2. Formuły wyznaczające ścieżki – każda ścieżka spełniająca formułę nazywana jest *ścieżką wyznaczoną przez tę formułę*.

a) formuła postaci $L(x)$ wyznacza ścieżki o długości 1:

Osoba(x), wyznacza zbiór {1, 7}.

b) formuły o postaci $L_1. \dots . L_n(x_1. \dots . x_n)$ i $L_1. \dots . L_n(v)$ wyznaczają ścieżki o długości n :

Osoba.Nazwisko($x.y$), wyznacza zbiór {(1, 2), (7, 8)},

to samo znaczenie ma formuła Osoba.Nazwisko(v).

c) formuła o postaci $P?.P'(v)$ wyznacza ścieżki, gdzie między fragmentem wyznaczonym przez P i P' występuje dokładnie jeden element:

(Osoba?.Miasto)(v), wyznacza zbiór {(1, 3, 4)},

to samo znaczenie ma formuła $Osoba(x) \wedge Miasto(y) \wedge (\{x\}?.\{y\})(v)$.

d) formuła o postaci $P*.P'(v)$ wyznacza ścieżki, gdzie między fragmentem wyznaczonym przez P i P' występuje dowolna liczba elementów (w tym może nie występować żaden):

(Osoba*.Miasto)(v), wyznacza zbiór {(1,3,4), (1,6,7,9), (7,9), (7,11,1,3,4)},

to samo znaczenie ma formuła $Osoba(x) \wedge Miasto(y) \wedge (\{x\}*. \{y\})(v)$.

Zauważmy, że ze względu na istniejący cykl można by się spodziewać, że formuła d) wyznacza również ścieżki:

(1, 6, 7, 11, 1, 3, 4), (1, 6, 7, 11, 1, 6, 7, 9), ...

co mogłoby prowadzić do nieskończonego zbioru ścieżek o dowolnej długości. W modelu PLO jest to jednak niemożliwe. Założyliśmy bowiem, że w ścieżce identyfikator może wystąpić co najwyżej jeden raz.

1. Definiowanie warunków spójności bazy danych:

„Jeśli dla osoby x istnieje jej podwładny z , to x jest kierownikiem dla z .”

$\forall x (\exists y, z \text{ Osoba.Podwładny?}(x.y.z) \Rightarrow \exists v \text{ Osoba.Kierownik?}(z.v.x))$.

Formuła ta jest prawdziwa dla następującego wartościowania zmiennych (ścieżek):

[$x/7, y/11, z/1, v/6$].

2. Formułowanie zapytań:

a) „Podaj nazwiska osób mieszkających na ulicy Polnej”:

$\{x.y \mid \text{Osoba.Nazwisko}(x.y) \wedge \exists z \text{ Osoba*.Ulica}(x.z) \wedge z = \text{„Polna”}\}$

Do odpowiedzi należy ścieżka (1, 2), co stanowi oczekiwaną odpowiedź, ale należy do niej także ścieżka (7, 8) - zmienna z oznacza wówczas ścieżkę (11, 1, 2), a więc także doprowadza do ulicy o nazwie „Polna”. Poprawny zapis polecenia może mieć postać:

$\{x.y \mid \text{Osoba.Nazwisko}(x.y) \wedge$

$\exists z (\text{Osoba.Adres.Ulica}(x.z) \vee \text{Osoba.Ulica}(x.z)) \wedge z = \text{„Polna”}\}$.

Przyjmując konwencję SQL-podobną, zapytanie to może zapisać w postaci:

```
SELECT    x.y
FROM      Osoba:x.Nazwisko:y,
          Osoba:o1.Adres.Ulica:z1 UNION Osoba:o2.Ulica:z2
WHERE (x = o1 AND z1 = „Polna”) OR (x = o2 AND z2 = „Polna”).
```

b) „Podaj osoby, których wszyscy podwładni mieszkają w Poznaniu”:

$\{x \mid \text{Osoba}(x) \wedge \forall v(\text{Osoba.Podwładny}^*. \text{Miasto}(x.v) \Rightarrow v = \text{„Poznań”})\}.$

4.2. SEMANTYKA JĘZYKA PATHCAL

Do zdefiniowania semantyki języka *PathCal* zastosujemy metodę denotacyjną. Definiowanie semantyki języka rozpoczynamy od zdefiniowania jego preinterpretacji, tj. od określenia przyporządkowania typom języka zbiorów elementów z uniwersum - zbiory te mogą być w ogólnym przypadku nieskończone.

Definicja 4.7.

Preinterpretacją nazywamy funkcję *dom*, która typom języka przyporządkowuje zbiory elementów z uniwersum obiektowo-ścieżkowego. Funkcja ta określona jest następująco:

$\text{dom}(U) = \mathbf{D},$
 $\text{dom}(OID) = \mathbf{OID},$
 $\text{dom}(PATH) = \mathbf{P},$
 $\text{dom}(\{OID\}) = \{X \mid X \subseteq \text{dom}(OID)\},$
 $\text{dom}(\{PATH\}) = \{X \mid X \subseteq \text{dom}(PATH)\}.$

Przypisywanie zmiennym wartości odbywa się za pomocą *wartościowania*. Każdej zmiennej typu τ przypisywane są przy tym elementy ze zbioru $\text{dom}(\tau)$.

Definicja 4.8.

Niech $\mathbf{X}$ oznacza zbiór wszystkich zmiennych języka *PathCal*.

Wartościowaniem ω zmiennych nazywamy funkcję z $\mathbf{X}$ do $\mathbf{D} \cup \mathbf{OID} \cup \mathbf{P}$, tj.

$\omega : \mathbf{X} \rightarrow \mathbf{D} \cup \mathbf{OID} \cup \mathbf{P}.$

Przez $\omega[x/c]$ oznaczać będziemy wartościowanie, które zmiennej $x \in \mathbf{X}$ przypisuje stałą c , a pozostałym zmiennym ze zbioru $\mathbf{X}$ przypisuje te same wartości co wartościowanie ω . Wartościowania takie nazywane są *podstawieniami*.

Zbiór wszystkich wartościowań oznaczać będziemy symbolem Ω , tj.

$\Omega = [\mathbf{X} \rightarrow \mathbf{D} \cup \mathbf{OID} \cup \mathbf{P}].$

W kolejnym kroku definiujemy funkcję zwaną *interpretacją*, która symbolom specyficznym języka *PathCal* (tj. stałym i etykietom) przypisuje odpowiednio stałe i skończone podzbiory zbioru **OID**.

Definicja 4.9.

Interpretacją języka PathCal nazywamy funkcję I określoną następująco:

1. Jeśli c jest stałą typu τ , to $I(c) \in \text{dom}(\tau)$.
2. Stałej funkcyjnej val przyporządkowana jest funkcja val określona w uniwersum, $I(val) = val$.
3. Symbolowi operacji kropkowej „.” na ścieżkach przypisujemy symbol złożenia ścieżek $\parallel$, tj. $I(.) = \parallel$.
4. Symbolowi operacji kropkowej „.” na zbiorach ścieżek przypisujemy symbol złożenia zbiorów ścieżek $\bullet$; podobnie dla symboli $?$ i $*$.
5. Jeśli L jest etykietą, $L \in \mathbf{L}$, to $I(L)$ jest skończonym podzbiorem zbioru **OID**. Zbiór $I(L)$ nazywamy *ekstensją* etykiety L , $\text{ext}(L)$. Ponadto zakładamy, że:
 - zbiory przypisane różnym etykietom są rozłączne

$$I(L) \cap I(L') = \emptyset, \text{ dla } L \neq L';$$
 - nie każdy identyfikator z **OID** musi być przypisany którejś z etykiet (etykietowanie częściowe).

■

Korzystając z pojęcia wartościowania zmiennych i interpretacji symboli specyficznych, a także z dziedzin określonych w preinterpretacji, możemy zdefiniować semantykę języka *PathCal*. Mówiąc o semantyce mamy na myśli odwzorowanie przypisujące każdemu termowi pewien element lub zbiór elementów z ustalonego uniwersum interpretacji tego języka. Wartość przypisana wyrażeniu złożonemu jest przy tym funkcją wartości wyrażeń składowych.

W przypadku formuł mówimy o ich *spełnianiu* przy ustalonej interpretacji i danym wartościowaniu.

Definicja 4.10.

Niech Ω będzie zbiorem wartościowań zmiennych języka *PathCal* i niech $\omega \in \Omega$. *Semantyką* termów języka *PathCal* nazywamy funkcję semT , która każdemu termowi p przy wartościowaniu ω przypisuje wartość $\text{semT}(p)(\omega)$ zdefiniowaną następująco:

1. Jeśli c jest stałą, to

$$\text{semT}(c)(\omega) = \text{semT}(c) = I(c).$$
2. Jeśli x jest zmienną, to

$$\text{semT}(x)(\omega) = \omega(x).$$
3. Jeśli L jest etykietą (predykatem 1-członowym), to

$$\text{semT}(L)(\omega) = \text{semT}(L) = I(L).$$
4. Jeśli $\{p_1, \dots, p_n\}$ jest termem typu $\{ID\}$, to

$$\text{semT}(\{p_1, \dots, p_n\})(\omega) = \{\text{semT}(p_1)(\omega), \dots, \text{semT}(p_n)(\omega)\}.$$

5. Jeśli p i p' są termami typu $PATH$, to

$$semT(p.p')(ω) = semT(p)(ω) \parallel semT(p')(ω).$$
6. Jeśli P i P' są termami typu $\{PATH\}$, to

$$semT(P.P')(ω) = semT(P)(ω) \bullet semT(P')(ω),$$

$$semT(P?)(ω) = semT(P)(ω)?$$

$$semT(P*)(ω) = semT(P)(ω)*.$$
7. Jeśli $q = \{p \mid \varphi\}$ jest zapytaniem, to

$$semT(\{p \mid \varphi\})(\Omega) = \{semT(p)(\omega) \mid \omega \in \Omega \wedge (I, \omega) \models \varphi\},$$
 przy czym $(I, \omega) \models \varphi$ jest relacją spełniania formuły φ przy interpretacji I i wartościowaniu ω . Relacja ta określona jest w definicji 4.11.

■

Definicja 4.11.

Niech I będzie interpretacją języka *PathCal*, ω niech będzie wartościowaniem zmiennych tego języka, $\omega \in \Omega$. Wówczas relacja spełniania, $\models$, przy interpretacji I i wartościowaniu ω zdefiniowana jest następująco:

1. $(I, \omega) \models L(p) \iff semT(p)(\omega) \in I(L);$
2. $(I, \omega) \models val(p) = v \iff val(semT(p)(\omega)) = semT(v)(\omega);$
3. $(I, \omega) \models P(p) \iff semT(p)(\omega) \in semT(P)(\omega);$
4. $(I, \omega) \models p = a \iff val(last(semT(p)(\omega))) = I(a);$
5. $(I, \omega) \models p = p' \iff val(last(semT(p)(\omega))) = val(last(semT(p')(\omega)));$
6. $(I, \omega) \models p = p' \iff last(semT(p)(\omega)) = last(semT(p')(\omega));$
7. $(I, \omega) \models p \in p' \iff last(semT(p)(\omega)) \in val(last(semT(p')(\omega)));$
8. $(I, \omega) \models p \subseteq p' \iff val(last(semT(p)(\omega))) \subseteq val(last(semT(p')(\omega)));$
9. $(I, \omega) \models \neg \varphi \iff \text{nieprawda, że } (I, \omega) \models \varphi;$
10. $(I, \omega) \models \varphi \wedge \varphi' \iff (I, \omega) \models \varphi \text{ oraz } (I, \omega) \models \varphi';$
11. $(I, \omega) \models \varphi \vee \varphi' \iff (I, \omega) \models \varphi \text{ lub } (I, \omega) \models \varphi';$
12. $(I, \omega) \models \varphi \Rightarrow \varphi' \iff \text{jeśli } (I, \omega) \models \varphi, \text{ to } (I, \omega) \models \varphi';$
13. $(I, \omega) \models \exists x \varphi \iff \text{istnieje stała } c \text{ taka, że } (I, \omega[x/c]) \models \varphi;$
14. $(I, \omega) \models \forall x \varphi \iff \text{dla każdej stałej } c, (I, \omega[x/c]) \models \varphi.$

■

Stałe atomowe wykorzystywane są jako wartości obiektów atomowych. Zapisujemy je w postaci łańcuchów znaków (ujętych w cudzysłowy) lub jako liczby, daty, kwoty w złotych itp. Ponieważ zakładamy, że wartości obiektów tego samego rodzaju mogą mieć różne typy standardowe, zatem komputerowa reprezentacja tych wartości musi uwzględniać możliwość rozpoznawania ich typów, również w przypadku wykonywania na nich operacji. Każda stała atomowa jest termem atomowym (typu U).

Przykładami stałych atomowych są:

„Jan Kowalski”, „100”, 100, 1000,00 zł, 1998.11.28.

Stałe identyfikacyjne utożsamiamy z identyfikatorami obiektów. Zapisujemy je jako liczby naturalne, przy czym z kontekstu zawsze jednoznacznie wynika czy konkretna liczba oznacza identyfikator, czy stałą atomową. Zmienne identyfikacyjne oznaczamy symbolami $x, y, z, \dots$ itp. niekiedy z indeksami i/lub apostrofami. Jeśli (1, „Kowalski”) jest obiektem, to 1 jest stałą identyfikacyjną (identyfikatorem obiektu). Zmienna x może oznaczać identyfikator 1, a więc także obiekt o tym identyfikatorze.

Formuły atomowe $\text{Osoba}(i)$ i $\text{Osoba}(x)$ są prawdziwe wówczas, gdy identyfikator i (aktualna wartość x) należy do ekstensji etykiety Osoba , tj.

$$i \in I(\text{Osoba}), \omega(x) \in I(\text{Osoba}).$$

Mówimy wówczas, że obiektowi o identyfikatorze i przypisano etykietę L .

Wyrażenia:

$\text{Osoba.Adres.Miasto}(1.3.4)$, $\text{Osoba.Adres.Miasto}(x.y.z)$, $\text{Osoba.Adres.Miasto}(v)$ są formułami atomowymi, gdzie $\text{Osoba.Adres.Miasto}$ jest termem typu $\{PATH\}$, a 1.3.4, $x.y.z$ i v są termami typu $PATH$.

Formuła

$$\text{Osoba.Adres.Miasto}(1.3.4)$$

jest prawdziwa wtedy i tylko wtedy, gdy prawdziwa jest formuła:

$$\text{Osoba}(1) \wedge \text{Adres}(3) \wedge \text{Miasto}(4) \wedge 3 \in \text{val}(1) \wedge 4 \in \text{val}(2).$$

Relacje równości.

W języku *PathCal* wprowadzamy dwie relacje równości, których argumentami są ścieżki:

- relację „=” oznaczającą równość wartości (stanów) obiektów, których identyfikatory zajmują ostatnią pozycję w porównywanych ścieżkach;
- relację „= =” oznaczającą identyczność identyfikatorów zajmujących ostatnie pozycje w porównywanych ścieżkach.

Przykład 4.6.

Niech dane będą obiekty częściowo etykietowane:

(1, A, {3, 4}), (2, B, {3, 4, 5}), (6, F, {3, 4}), (3, „a”), (4, „b”), (5, „a”).

Wówczas:

- formuła $A(x) \wedge F(y) \wedge x = y$ jest prawdziwa przy wartościowaniu $[x/1, y/6]$, gdyż $\text{val}(x) = \{3, 4\} = \text{val}(y)$; przy tym samym wartościowaniu fałszywa jest jednak formuła $A(x) \wedge F(y) \wedge x = = y$, gdyż $1 \neq 6$;
- formuła $A?(v) \wedge B?(w) \wedge v = w$ jest prawdziwa dla wartościowań:
- $[v/(1, 3), w/(2, 3)], [v/(1, 4), w/(2, 4)], [v/(1, 3), w/(2, 5)]$;
- formuła $A?(v) \wedge B?(w) \wedge v = = w$ jest prawdziwa dla wartościowań:
- $[v/(1, 3), w/(2, 3)], [v/(1, 4), w/(2, 4)]$.



Relacje zawierania i przynależności.

Relacje te określone są na ścieżkach. Relacje zawieranie, $p \subseteq p'$, spełniona jest wtedy, gdy stan obiektu, którego identyfikator zajmuje ostatnią pozycję w p jest zawarty w stanie obiektu, którego identyfikator zajmuje ostatnią pozycję w p' . Relacje przynależności, $p \in p'$, spełniona jest wtedy, gdy identyfikator zajmujący ostatnią pozycję w p jest zawarty w stanie obiektu, którego identyfikator zajmuje ostatnią pozycję w p' .

Przykład 4.7.

Rozważmy zbiór obiektów z przykładu 4.6. Wówczas:

- a) formuła $A(x) \wedge B(y) \wedge x \subseteq y$ jest prawdziwa przy wartościowaniu $[x/1, y/2]$; jest ona równoważna formule $A(x) \wedge B(y) \wedge \forall z (z \in x \Rightarrow z \in y)$;
- b) formuła $\exists z (A(x) \wedge z \in x)$ jest prawdziwa dla wartościowań $[x/1, z/3]$ i $[x/1, z/4]$. ■

5. SYSTEM BAZY DANYCH OBIEKTÓW CZĘŚCIOWO ETYKIETOWANYCH

W teorii baz danych można wyróżnić dwa podejścia: podejście teorio-dowodowe i podejście teorio-modelowe [26]. W podejściu teorio-dowodowym baza danych rozumiana jest jako zbiór zdań języka bazy danych (zbiór faktów), a udzielanie odpowiedzi realizowane jest za pomocą mechanizmów wnioskowania (dowodzenia twierdzeń). Podejście takie jest charakterystyczne dla dedukcyjnych baz danych oraz dla baz danych budowanych w oparciu o Prolog. System bazy danych odpowiada wówczas pojęciu *teorii* w rozumieniu logiki, a więc składa się z języka, zbioru aksjomatów (faktów tworzących bazę danych) i reguł wnioskowania.

W podejściu teorio-modelowym zakłada się istnienie skończonego *modelu*, względem którego wartościowane są wyrażenia *języka bazy danych* [11, 23]. Model ten z formalnego punktu widzenia jest pewną strukturą matematyczną, a w praktyce stanowi zbiór danych zapamiętanych w pamięci komputera (najczęściej zewnętrznej) – odpowiada więc tradycyjnemu rozumieniu stanu bazy danych jako zbioru różnorodnych struktur danych. Związek wyrażeń języka bazy danych i elementów modelu ustala *semantyka* języka – pojęciom indywidualnym przyporządkowywane są indywidualne elementy modelu, pojęciom zbiorowym ich zbiory, a wyrażeniom złożonym operacje na elementach przypisanych wyrażeniom składowym. Takie rozumowanie wykorzystywane jest zarówno do obliczania wyrażeń (w szczególności

do wyznaczania odpowiedzi na zapytania), jak i do stwierdzania prawdziwości lub fałszywości formuł.

Ważną rolę w każdym systemie bazy danych pełnią *warunki spójności* (zwane także *więzami integralności* lub *ograniczeniami semantycznymi*). Jest to wyróżniony zbiór formuł języka bazy danych, o których zakłada się, że są prawdziwe w każdym stanie bazy danych. Również więc w tym sensie stan bazy danych jest *modelem zbioru warunków spójności* – modelem zbioru formuł (w myśl teorii modeli) jest bowiem każda taka struktura, w której prawdziwe są wszystkie formuły z tego zbioru.

Definicja 5.1 precyzuje rozumienie systemu bazy danych w ujęciu teorio-modelowym.

Definicja 5.1.

Systemem bazy danych nazywamy czwórkę:

$$SBD = (\text{JęzykBD}, \text{WarSpój}, \text{BD}, \text{FunkSem}),$$

gdzie:

1. *JęzykBD* – język bazy danych; do definiowania zarówno baza danych, jej warunków spójności, jak również zapytań względem tej bazy danych.
2. *WarSpój* – warunki spójności bazy danych; zbiór formuł języka bazy danych, o którym zakłada się, że są prawdziwe w każdym stanie bazy danych.
3. *BD* – stan (wystąpienie) baza danych; skończony zbiór danych stanowiący model (w sensie logicznym) zbioru warunków spójności.
4. *FunkSem* – funkcje semantyczne; odwzorowania przypisujące znaczenie (semantykę) wyrażeniom języka bazy danych, pozwalają obliczać wartości i stwierdzać prawdziwości wyrażań języka bazy danych.



W systemie bazy danych obiektów częściowo etykietowanych, przyjmujemy następującą interpretację składowych ogólnie rozumianego systemu bazy danych:

1. Językiem bazy danych jest *PathCal* – język ten w dalszym ciągu rozszerzymy w taki sposób, aby tworzone w nim odpowiedzi na zapytania mogły być z powrotem traktowane jako elementy bazy danych.
2. Warunki spójności specyfikowane są jako formuły języka *PathCal*. Rola tych warunków w rozważanej klasie systemów baz danych bezschematowych jest szczególnie istotna. Nie ma bowiem schematu, który nakładałby ograniczenia na dopuszczalne stany bazy danych. Uzyskujemy dzięki temu dużą dowolność w strukturalizowaniu danych, ale stwarza to niebezpieczeństwo generowania struktur wewnętrznie sprzecznych (nie-spójnych). Problem właściwego podejścia do warunków spójności w bezschematowych bazach danych jest ciągle na etapie intensywnych badań [22]. Przyjmujemy, że do zbioru warunków spójności należą wszystkie formuły definiujące obiekty częściowo etykietowane, a więc formuły o postaci:

$$L(i) \wedge val(i) = a - \text{definicja etykietowanego obiektu atomowego,}$$

$val(i) = a$ – definicja nieetykietowanego obiektu atomowego,

$L(i) \wedge val(i) = \{i_1, \dots, i_n\}, n \geq 1$, definicja etykietowanego obiektu złożonego,

$val(i) = \{i_1, \dots, i_n\}, n \geq 1$, definicja nieetykietowanego obiektu złożonego,

oraz inne formuły zamknięte (zdania) stwierdzające istnienie pewnych obiektów lub istnienie pewnych zależności między obiektami.

3. Baza danych składa się ze skończonego zbioru obiektów tworzących spójny i skończony podzbiór uniwersum obiektowo-ścieżkowego.

4. Funkcje semantyczne obejmują funkcję interpretacji I , funkcję $semT$ wyznaczającą wartości termów oraz relację spełniania $\models$ pozwalającą określić prawdziwość formuł.

Następujący przykład ilustruje teorio-modelowe podejście do systemu bazy danych obiektów częściowo etykietowanych.

Przykład 5.1.

Przypuśćmy, że chcemy aby zbiór obiektów etykietowanych przedstawiony na rysunku 5.1. był bibliograficzną bazą danych.

1. Biblio {2,10}
2. Dok {3, 4, 9}
3. Tytuł „Active Database Systems”
4. Redaktor {5, 6, 7, 8}
5. Imię „Jennifer”
6. Nazwisko „Widom”
7. Imię „Stefano”
8. Nazwisko „Ceri”
9. Rok 1996
10. Dok {11, 12, 13, 14}
11. Tytuł „Introduction to Active Database Systems”
12. Autor {7, 8}
13. Autor {5, 6}
14. Źródło {2}

Rys. 5.1. Zbiór obiektów etykietowanych
tworzący bibliograficzną bazę danych

1. Językiem bibliograficznej bazy danych będzie język *PathCal*, przy czym zbiorem etykiet jest w tym przypadku zbiór:

$L = \{\text{Biblio, Dok, Tytuł, Redaktor, Imię, Nazwisko, Rok, Autor, Źródło}\}.$

2. Zbiór warunków spójności bazy danych składa się z dwóch części:

a) Zbiór formuł (aksjomatów) definiujący bazę danych. Obliczenie tych formuł daje w wyniku bazę danych z rys. 5.1:

$$\text{Biblio}(1) \wedge \text{val}(1) = \{2, 10\},$$

$$\text{Dok}(2) \wedge \text{val}(2) = \{3, 4, 9\},$$

$$\text{Dok}(10) \wedge \text{val}(10) = \{11, 12, 13, 14\},$$

$$\text{Tytuł}(3) \wedge \text{val}(3) = \text{„Active Database Systems”},$$

$$\text{Tytuł}(11) \wedge \text{val}(11) = \text{„Introduction to Active Database Systems”},$$

$$\text{Redaktor}(4) \wedge \text{val}(4) = \{5, 6, 7, 8\},$$

$$\text{Imię}(5) \wedge \text{val}(5) = \text{„Jennifer”},$$

$$\text{Imię}(7) \wedge \text{val}(7) = \text{„Stefano”},$$

$$\text{Nazwisko}(6) \wedge \text{val}(6) = \text{„Widom”},$$

$$\text{Nazwisko}(8) \wedge \text{val}(8) = \text{„Ceri”},$$

$$\text{Autor}(12) \wedge \text{val}(12) = \{7, 8\},$$

$$\text{Autor}(13) \wedge \text{val}(13) = \{5, 6\},$$

$$\text{Rok}(9) \wedge \text{val}(9) = 1996,$$

$$\text{Źródło}(14) \wedge \text{val}(14) = \{2\}.$$

b) Zależności między danymi. Wymieńmy kilka warunków, które powinny być zawsze spełnione w rozważanej bazie danych:

- każdy dokument musi mieć tytuł:

$$\forall x (\text{Dok}(x) \Rightarrow \exists y \text{Dok.Tytuł}(x.y));$$

- każdy dokument ma redaktora lub autora:

$$\forall x (\text{Dok}(x) \Rightarrow \exists y (\text{Dok.Autor}(x.y) \vee \text{Dok.Redaktor}(x.y)));$$

- jeśli dokument ma podane źródło, to źródłem tym jest dokument mający redaktora:

$$\forall x (\text{Dok}(x) \wedge \exists y \text{Dok.Źródło}(x.y) \Rightarrow \exists z, v \text{Dok.Źródło.Dok.Redaktor}(x.y.z.v)).$$

3. Interpretacja I :

a) dla etykiet:

$$I(\text{Biblio}) = \{1\},$$

$$I(\text{Dok}) = \{2, 10\},$$

$$I(\text{Tytuł}) = \{3, 11\},$$

$$I(\text{Redaktor}) = \{4\},$$

$$I(\text{Imię}) = \{5, 7\},$$

$$I(\text{Nazwisko}) = \{6, 8\},$$

$$I(\text{Autor}) = \{12, 13\},$$

$$I(\text{Rok}) = \{9\},$$

$$I(\text{Źródło}) = \{14\},$$

b) dla symbolu funkcyjnego val : $I(\text{val})(i) = \text{val}(i).$

Interpretacja I wyznacza wystąpienie **I** bazy danych:

$$\mathbf{I} = I(\mathbf{L}) = (\cup \{I(L) \mid L \in \mathbf{L}\}),$$

a więc składa się ze zbioru ekstensji etykiet. Domykając ten zbiór ze względu na operacje tworzenia ścieżek, otrzymujemy skończone uniwersum obiektowo ścieżkowe, w którym można zidentyfikować następujące ścieżki:

– ścieżki o długości 1 (jest ich tyle ile identyfikatorów plus ścieżka złożona z identyfikatora zerowego nil):

nil, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19.

– wybrane ścieżki rozpoczynające się od identyfikatora 1:

(1, 2), (1, 2, 3), (1, 2, 4, 5), (1, 11, 19, 2, 7, 9), ...

Znaczenie funkcji *semT* i relacji spełniania $\models$ jest takie jak zdefiniowaliśmy przy omawianiu semantyki języka *PathCal* (def. 3.6 i 3.7). Zdefiniowane pojęcia możemy więc wykorzystać do obliczania zapytań formułowanych względem bibliograficznej bazy danych.

Rozważmy następujące zapytanie względem bazy danych z przykładu 5.1.

Zapytanie 5.1.

„Podaj pozycje bibliograficzne, których autorem jest Ceri”

$\{b.d.t \mid \exists a, n (\text{Biblio.Dok.Tytuł}(b.d.t) \wedge \text{Dok.Autor.Nazwisko}(d.a.n) \wedge n = \text{„Ceri”})\}$.

W zapytaniu 5.1 występują trzy zmienne wolne: *b*, *d* i *t*. Zgodnie z zasadą wartościowania zapytań, obliczamy je następująco:

$$\text{semT}(\{b.d.t \mid \varphi\})(\Omega) = \{\omega(b.d.t) \mid \omega \in \Omega \wedge (I, \omega) \models \varphi\},$$

gdzie

b.d.t jest termem ścieżkowym,

φ jest formułą:

$$\exists a, n (\text{Biblio.Dok.Tytuł}(b.d.t) \wedge \text{Dok.Autor.Nazwisko}(d.a.n) \wedge n = \text{„Ceri”}).$$

Wyznamy najpierw zbiór wartościowań Ω . W ogólnym przypadku jest to zbiór nieskończony. W rozważanym zapytaniu zakresy zmiennych *b*, *d* oraz *t*, są ograniczone za pomocą formuły *Biblio.Dok.Tytuł*(*b.d.t*). Podobnie dla pozostałych zmiennych. Rozważania możemy więc ograniczyć do *aktywnej dziedziny* zapytania. Wówczas zakresy *ext*() dla poszczególnych zmiennych są następujące:

$$\text{ext}(b) = \{1\}, \quad \text{ext}(d) = \{2, 10\}, \quad \text{ext}(t) = \{3, 11\},$$

$$\text{ext}(a) = \{12, 13\}, \quad \text{ext}(n) = \{6, 8\}.$$

Zbiór Ω wszystkich wartościowań zawiera więc *N* elementów, gdzie *N* jest iloczynem liczebności zbiorów stanowiących zakresy dla poszczególnych zmiennych. W rozważanym przypadku *N* = 16.

Uwzględnienie warunku *n* = „Ceri” pozwala na redukcję zbioru *ext*(*n*) do zbioru {8}. Dalej, jeśli uwzględnimy formułę *Dok.Autor.Nazwisko*(*d.a.n*), w myśl której musi zachodzić warunek *n* ∈ *val*(*a*), to zbiór *ext*(*a*) zostanie zredukowany do zbioru {12}.

Wówczas zbiór Ω zawiera cztery elementy wyszczególnione w poniższej tabeli:

Ω :

ω	d	t	b	a	N
ω_1	2	3	1	12	8
ω_2	2	11	1	12	8
ω_3	10	3	1	12	8
ω_4	10	11	1	12	8

Widać, że kwalifikator zapytania

$\varphi = \exists a, n \text{ (Biblio.Dok.Tytuł}(b.d.t) \wedge \text{Dok.Autor.Nazwisko}(d.a.n) \wedge n = \text{„Ceri”}$
spełniony jest jedynie dla wartościowania ω_4 .

Odpowiedź na zapytanie jest więc zbiorem zawierającym jedną ścieżkę

$\{(1, 10, 11)\}$, co odpowiada strukturze obiektów:

- 1 Biblio {10}
- 10 Dok {11}
- 11 Tytuł „Introduction to Active Database Systems”



6. STRUKTURALIZOWANIE ODPOWIEDZI NA ZAPYTANIA

6.1. PROBLEM WŁĄCZANIA WYNIKÓW ZAPYTAŃ DO BAZY DANYCH

Ważną cechą systemów baz danych jest to, że odpowiedź na zapytanie może być powoływana w kolejnych zapytaniach. Z punktu widzenia operowania w bazie danych odpowiedź taka winna więc być traktowana jak element bazy danych. W standardowych systemach baz danych mechanizmem umożliwiającym takie traktowanie odpowiedzi jest mechanizm *perspektyw* (ang. *view*). Perspektywa jest bytem wirtualnym, który nie musi istnieć fizycznie w bazie danych, ale ma określoną nazwę i w związku z tym może być powoływany w zapytaniach jak każdy inny byt. Fizyczne włączanie perspektyw do bazy danych nosi nazwę *materializacji perspektyw*. Zauważmy, że z formalnego punktu widzenia możliwość tworzenia perspektyw i nadawania im nazw wyprowadza poza język bazy danych, gdyż pojawia się konieczność powiększania wyjściowego zbioru nazw relacji.

Z podobną trudnością mamy do czynienia w modelu *PLO*. Trudność ta jest spotęgowana faktem, że odpowiedź często trzeba traktować jako tworzenie nowych obiektów, a z tym związane jest tworzenie (wybieranie) nowych identyfikatorów. Rozważmy następujące możliwości:

- a) obiekt należący do odpowiedzi nie otrzymuje nowego identyfikatora – dziedziczy go od swego obiektu macierzystego, tj. od tego, z którego został utworzony;
- b) obiekt należący do odpowiedzi otrzymuje nowy identyfikator przy pozostawieniu etykiety obiektu macierzystego;
- c) obiekt należący do odpowiedzi otrzymuje nowy identyfikator i nową etykietę;
- d) identyfikowanie i etykietowanie obiektów specyfikowane jest w wyrażeniach języka.

Zanim przejdziemy do formalnych definicji, przeanalizujemy problemy związane z włączaniem odpowiedzi do bazy danych. Rozważmy powyższe trzy warianty postępowania z uzyskaną odpowiedzią $\{(1, 10, 11)\}$ na zapytania 5.1.

Wariant 1. Dziedziczenie identyfikatorów od obiektów macierzystych

Do bazy danych wprowadzamy obiekty:

1	Biblio	{10}
10	Dok	{11}
11	Tytuł „Introduction to Active Database Systems”.	

Jest to niepoprawne, gdyż prowadziłoby do występowania w bazie danych dwóch obiektów o tych samych identyfikatorach i różnych stanach: $(1, \text{Biblio}, \{2, 10\})$ i $(1, \text{Biblio}, \{10\})$ - *anomalia sprzecznych obiektów*.

Wariant 2. Nadawanie nowych identyfikatorów i starych etykiet

Do bazy danych wprowadzamy obiekty:

15	Biblio	{16}
16	Dok	{17}
17	Tytuł „Introduction to Active Database Systems”.	

Zauważmy, że ponowne wykonanie zapytania 5.1 względem tak poszerzonej bazy danych dałoby w odpowiedzi zbiór dwóch ścieżek: $\{(1, 10, 11), (15, 16, 17)\}$, co jest niedopuszczalne. Nazwiemy to *anomalią powtórnego wykonania zapytania*.

Wariant 3. Nadawanie nowych identyfikatorów i nowych etykiet

Do bazy danych wprowadzamy obiekty:

15	Biblio1	{16}
16	Dok1	{17}
17	Tytuł1 „Introduction to Active Database Systems”.	

Wadą tego rozwiązania jest niepotrzebne powielenie obiektu atomowego $(11, \text{Tytuł}, \text{„Introduction to Active Database Systems”})$.

Jest to *anomalia duplikowania obiektów*.

Wariant 4. Sterowanie określaniem identyfikatorów i etykiet

Analiza powyższych anomalii prowadzi do wniosku, że najlepszym rozwiązaniem byłoby interpretowanie odpowiedzi na zapytanie 5.1 jako zbioru obiektów:

```

15      Biblio1 {16}
16      Dok1 {11}
11      Tytuł    „Introduction to Active Database Systems”.

```

Niektóre obiekty otrzymują więc nowe identyfikatory i tym samym nowe etykiety, a niektóre pozostają identyczne z obiektami macierzystymi.

Rozważania powyższe prowadzą do wniosku, że jeśli chcemy traktować odpowiedź jako zbiór obiektów wprowadzanych do bazy danych, to musimy określić które z obiektów są:

- nowo tworzonymi z obiektów macierzystych;
- identyczne z obiektami macierzystymi.

Dla obiektów nowo tworzonych musi być ponadto zdefiniowany sposób przypisywania im nowych identyfikatorów, nowych etykiet oraz sposób określania ich stanów.

6.2. REGUŁY W JĘZYKU PATHLOG

Potrzeba rozszerzenia języka *PathCal* do języka regułowego wynika z konieczności określania postaci danych tworzących odpowiedzi na zapytania. Odpowiedź w języku *PathCal* jest zbiorem ścieżek, przy czym ścieżki należące do tego zbioru mogą być niejednorodne (mogą mieć na przykład różną długość). Po to, aby wynik zapytania mógł być ponownie uwzględniany przy obliczaniu kolejnych zapytań, musi być zapamiętany w bazie danych jako (być może tymczasowy) zbiór obiektów. Ze zbioru ścieżek tworzących odpowiedzi na zapytanie należy więc wygenerować zbiór obiektów równoważny temu zbiorowi ścieżek..

W tym podrozdziale proponujemy język regułowy o nazwie *PathLog*, którego konstrukcje (a przede wszystkim głowy reguł) służą do sterowania procesem tworzenia wynikowego zbioru obiektów reprezentujących odpowiedzi na zapytanie. Z założenia obiekty te automatycznie traktowane są jako obiekty bazy danych. Włączenie tych obiektów do bazy danych zmienia jednocześnie system bazy danych, gdyż zmianie ulega jej język – w zbiorze etykiet (predykatów 1-członowych) pojawiają się nowe elementy.

Znaczenie głowy reguły w konstruowaniu nowych obiektów jest następujące:

- wpływa na sposób tworzenia obiektów odpowiedzi: jeśli występuje w niej nowa etykieta, to tworzony jest nowy obiekt; jeśli występuje etykieta już istniejąca w systemie, to nowy obiekt nie jest tworzony a odwołujemy się do obiektu już istniejącego w bazie danych;

- wpływa na sposób przypisywania etykiet – nowe obiekty przypisywane są nowym etykiatom.

W języku *PathLog*, podobnie jak w *Datalogu* [26], definiujemy *reguły* i *programy*, przy czym programy (ciągi reguł) mogą być *nierekurencyjne* lub *rekurencyjne* [14]. W niniejszej pracy ograniczamy się tylko do reguł. W dalszym ciągu mówiąc o *termie etykietowym* mamy na myśli wyrażenia (termy typu $\{PATH\}$) o postaci $L_1. \dots L_n$, gdzie $n \geq 1$, a każde L_k jest *etykietą*, która bądź istnieje w języku, bądź ma być przypisana nowo tworzonemu obiektowi (a więc tym samym ma być dołączona do alfabetu języka).

Definicja 6.1.

Niech $\{p \mid \varphi\}$ będzie zapytaniem języka *PathCal*, a H niech będzie termem etykietowym. O H zakładamy, że wszystkie zawarte w niej etykiety, za wyjątkiem co najwyżej ostatniej, nie występują w języku *PathCal* i że długość H nie przekracza długości najkrótszej ścieżki należącej do odpowiedzi. Wówczas wyrażenie:

$$H \leftarrow \{p \mid \varphi\}$$

nazywamy *regułą PathLogu*. H nazywamy *głową*, a $\{p \mid \varphi\}$ *ciałem* reguły. ■

Przykład 6.1.

Zapytanie 5.1 można zapisać w postaci reguły:

$$\begin{aligned} \text{Bibliol.Dokl.Tytuł} \leftarrow \{b.d.t \mid \exists a, n (\text{Biblio.Dok.Tytuł}(b.d.t) \wedge \\ \wedge \text{Dok.Autor.Nazwisko}(d.a.n) \wedge n = \text{„Ceri”})\}. \end{aligned}$$

Wynikowy zbiór obiektów:

15	Bibliol	{16}
16	Dokl	{11}
11	Tytuł	„Introduction to Active Database Systems”.

■

Algorytm 6.1. podaje sposób obliczania reguł języka *PathLog*.

Algorytm 6.1.

Wejście:

Baza danych BD oraz reguła $L_1. \dots L_n \leftarrow \{p \mid \varphi\}$.

Niech $L = \{L_1. \dots L_N\}$,

gdzie

$N = n - 1$, jeśli etykieta L_n występuje w języku *PathCal* i $N = n$ w przeciwnym razie.

Wyjście:

Funkcja I' wyznaczająca ekstensje etykiet ze zbioru L oraz funkcja val' wyznaczająca stany obiektów o identyfikatorach należących do ekstensji etykiet ze zbioru L . Nowa interpretacja będzie więc równa $I = I \cup I'$.

Kroki:

Przyjmij $I'(L_k) = \emptyset$, dla każdego $1 \leq k \leq N$.

Dla każdej ścieżki $p = (i_1, \dots, i_m)$ należącej do odpowiedzi, $m \geq N$, wykonuj:

dla każdego k , $1 \leq k \leq N$, wykonuj:

- obierz nowy identyfikator i'_k ;
- dołącz i'_k do zbioru $I'(L_k)$,
 $I'(L_k) = I'(L_k) \cup \{i'_k\}$,
- przyjmij
 $val(i'_k) = \emptyset$.
- jeśli $k > 1$, to dołącz i'_k do zbioru $val(i'_{k-1})$,
 $val'(i'_{k-1}) = val'(i'_{k-1}) \cup \{i'_k\}$.
- jeśli $k = N$ i $N = n - 1$, to dołącz i_n do zbioru $val(i'_N)$,
 $val'(i_n) = val'(i'_N) \cup \{i_n\}$;
- jeśli $k = N$ i $N = n$, to przyjmij
 $val'(i'_N) = val(i_n)$.



7. TRANSFORMACJA STRUKTUR DANYCH DO ZBIORU OBIEKTÓW CZĘŚCIOWO ETYKIETOWANYCH

W rozdziale 3 przedstawiliśmy reprezentację wybranych struktur danych za pomocą zbioru obiektów częściowo etykietowanych. Obecnie podamy algorytm transformacji struktur danych do zbioru obiektów w pełni lub częściowo etykietowanych.

7.1. TRANSFORMACJA WARTOŚCI ZŁOŻONYCH

Pokażemy jaką klasę struktur danych może być przedstawiona za pomocą modeli obiektów częściowo etykietowanych. Jako etykiety chcemy wykorzystywać tylko nazwy (atrybutów, klas, relacji) występujące w tych strukturach. Nie dopuszczamy natomiast wprowadzania ani nazw zewnętrznych, ani innych nazw „sztucznych”.

Najpierw zdefiniujemy klasę struktur danych (zwaną *wartościami złożonymi*) i pokażemy, w jaki sposób można ją reprezentować za pomocą rozważanej klasy obiektów częściowo etykietowanych.

Definicja 7.1.

Niech $\mathbf{D}$ będzie zbiorem stałych atomowych, a $\mathbf{A}$ niech będzie zbiorem atrybutów. Uniwersum wartości złożonych definiujemy następująco:

2. Każda stała atomowa $a \in \mathbf{D}$ jest *atomową* wartością złożoną.
3. Dla $n \geq 1$, niech $A_1, \dots, A_n$ będą różnymi atrybutami ze zbioru $\mathbf{A}$, a $e_1, \dots, e_n$ niech będą wartościami złożonymi. Wówczas zbiór par (pól) $[A_1 : e_1, \dots, A_n : e_n]$ nazywamy *krotkową* wartością złożoną.
4. Niech dla $n \geq 1$, $e_1, \dots, e_n$ będą wartościami złożonymi. Wówczas zbiór $\{e_1, \dots, e_n\}$ nazywamy *zbiorową* wartością złożoną.

■

Wartości złożone są więc tworzone rekurencyjnie. Przyjmujemy, że zadany zbiór stałych atomowych jest zbiorem wartości złożonych, a następnie domykamy go stosując rekurencyjnie konstruktor wartości krotkowych (przy zadanym zbiorze atrybutów) i konstruktor wartości zbiorowych.

Pokażemy, że zbiór wartości złożonych można przekształcić w zbiór obiektów częściowo etykietowanych. Przekształcenie to obejmuje kilka kroków:

3. Dany jest skończony zbiór wartości złożonych e (zbiorowa wartość złożona), której ma być nadana pewna nazwa N (nazwa ta jest elementem alfabetu języka danych).
4. Wspomniane przypisanie traktujemy jako tworzenie krotki $[N : e]$ złożonej z jednego pola, gdzie N jest nazwą pola (atrybutem), a e jest jego wartością.
5. Krotkę $[N : e]$ przekształcamy w multikrotkę (patrz definicja 7.2).
6. Multikrotkę przekształcamy w zbiór obiektów częściowo etykietowanych (algorytm 7.1).

Transformację zbioru wartości złożonych w zbiór obiektów częściowo etykietowanych opisuje algorytm 7.1. Pomocniczą rolę w tym algorytmie odgrywa operacja przekształcająca krotkę w *multikrotkę*, tj. w krotkę, w której może wystąpić kilka pól o tych samych nazwach.

Istota operacji tworzenia multikrotki polega na tym, że każde pole z atrybutem wielowartościowym (wartością atrybutu jest zbiór) zastępujemy zbiorem pól o tych samych nazwach i wartościach będących poszczególnymi elementami zbioru. Na przykład krotkę $[A : \{a, b, c\}]$ przekształcamy w multikrotkę $[A : a, A : b, A : c]$.

Poniższy przykład ukazuje problemy w przypadku występowanie „zbioru zbiorów”. Pokazuje jednocześnie, że pozostając na gruncie pełnego etykietowanie (co ma miejsce na przykład w modelu Tsimmis/OEM) nie jesteśmy w stanie reprezentować ważnej klasy struktur danych.

Przykład 7.1.

Przypuśćmy, że dla pewnej nazwy R , $I(R) = \{\{a, b\}, \{c, d\}\}$. Etykietę R przypisujemy zbiorom $\{a, b\}$ i $\{c, d\}$ (są one elementami zbioru niezagnieżdżonego, tj. o głębokości 0), a elementy a, b, c i d pozostają bez etykiet (a). Zauważmy, że „wejście” z etykietą R do zbiorów o głębokości większej niż 0 daje zbiór obiektów w pełni etykietowanych (b).

(a)

1 R $\{2, 3\}$
 2 a
 3 b
 4 R $\{5, 6\}$
 5 c
 6 d

(b)

1 R a
 2 R b
 3 R c
 4 R d

Reprezentacja (b) jest jednak nieodwracalna – tracimy informację o przynależności elementów do zbiorów. ■

Definicja 7.2.

Niech e będzie dowolną wartością złożoną. Funkcja $pmulti(e)$ przekształcająca e w multikrotkę częściowo etykietowaną zdefiniowana jest następująco:

$$pmulti(e) =$$

case e **of**

$$a : a,$$

$$[A_1 : e_1, \dots, A_n : e_n] : pmulti(A_1 : e_1) \cup \dots \cup pmulti(A_n : e_n),$$

$$\{e_1, \dots, e_n\} : \{pmulti(e_1), \dots, pmulti(e_n)\}.$$

endcase,

$$pmulti(f) =$$

case f **of**

$$A : a : [A : a],$$

$$A : [A_1 : e_1, \dots, A_n : e_n] : [A : pmulti([A_1 : e_1, \dots, A_n : e_n])],$$

$$A : \{e_1, e_2, \dots, e_n\} : pmulti(A : e_1) \cup \dots \cup pmulti(A : e_n),$$

$$A : \{e_1, e_2, \dots, e_n\} : [A : pmulti(\{e_1, e_2, \dots, e_n\})].$$

endcase

przy czym symbol $\{ \}$ oznacza zbiór o poziomie zagnieżdżenia 0, a $\{ \}$ – zbiór o poziomie zagnieżdżenia większym niż 0. ■

Algorytm 7.1. przekształca dowolną wartość złożoną w zbiór obiektów częściowo etykietowanych.

Algorytm 7.1.

Wejście: Nazwa R , której interpretacja I przypisuje zbiorową wartość złożoną $\{e_1, e_2, \dots, e_N\}$.

Wyjście: Zbiór obiektów częściowo etykietowanych reprezentujących wystąpienie $I(R)$.

Kroki:

1. Tworzymy krotkę $K = [R : \{e_1, e_2, \dots, e_N\}]$.
2. Krotkę K przekształcamy w multikrotkę, $M = pmulti(K) = [R : c_1, \dots, R : c_m]$.
3. Dla multikrotki $[R : c_1, \dots, R : c_m]$ stosujemy funkcję plo transformującą ją w zbiór obiektów częściowo etykietowanych. Funkcja plo określona jest następująco:

$plo([R : c_1, \dots, R : c_m]) = prep(newId(), R, c_1) \cup \dots \cup prep(newId(), R, c_m)$,

gdzie funkcja $newId()$ zwraca nowy identyfikator dla tworzonego obiektu

$prep(i, A, e) =$

case e of

$a : \{(i, A, a)\};$

$[A_1 : e_1, \dots, A_n : e_n] : \{(i, A, \{i_1 := newId(), \dots, i_n := newId()\})\} \cup$
 $\cup prep(i_1, A_1, e_1) \cup \dots \cup prep(i_n, A_n, e_n);$

$\{e_1, \dots, e_n\} : \{(i, A, \{i_1 := newId(), \dots, i_n := newId()\})\} \cup$
 $\cup prep'(i_1, e_1) \cup \dots \cup prep'(i_n, e_n);$

endcase

$prep'(i, e) =$

case e of

$a : \{(i, a)\};$

$[A_1 : e_1, \dots, A_n : e_n] : \{(i, \{i_1 := newId(), \dots, i_n := newId()\})\} \cup$
 $\cup prep(i_1, A_1, e_1) \cup \dots \cup prep(i_n, A_n, e_n);$

$\{e_1, \dots, e_n\} : \{(i, \{i_1 := newId(), \dots, i_n := newId()\})\},$
 $\cup prep'(i_1, e_1) \cup \dots \cup prep'(i_n, e_n);$

endcase

■

Przykład 7.2 pokazuje zastosowanie algorytmu 7.1 do transformacji nieregularnej wartości złożonej, zawierającej grupę powtarzalną (tablicę krotek), w zbiór obiektów częściowo etykietowanych.

Przykład 7.2.

Niech $I(\text{Temperatura}) = \{\{\text{Poznań}, \{[\text{Data} : 1998.12.19, \text{Wartość} : 5.5],$
 $[\text{Data} : 1998.12.20, \text{Wartość} : 7.0]\},$
 $\{\text{Warszawa}, [\text{Data} : 1998.12.19, \text{Wartość} : 4.0]\}\}.$

Stosując algorytm 7.1 otrzymujemy następującą reprezentację wystąpienia $I(\text{Temperatura})$ w postaci zbioru obiektów częściowo etykietowanych:

1	Temperatura	{2, 3}
2	Poznań	
3	{4, 5}	
4	{6, 7}	
6	Data	1998.12.19
7	Wartość	5.5

5	{8, 9}		
8	Data	1998.12.20	
9	Wartość	7.0	
10	Temperatura	{11, 12}	
11	Warszawa		
12	{13, 14}		
13	Data	1998.12.19	
14	Wartość	4.0	

7.2. TRANSFORMACJA OBIEKTÓW OPARTYCH NA IDENTYFIKATORACH

W modelu obiektowym opartym na identyfikatorach (*identity-based*) [6,8,9,17], przyjmuje się, że każdy obiekt jest parą (*oid*, *val*), gdzie *oid* jest niezmiennym identyfikatorem obiektu, a *val* jest wartością (lub *stanem*) obiektu. Silne typowanie wymaga zgodności typów identyfikatora i wartości. Żądamy mianowicie, aby wartość *val* była typu τ wtedy i tylko wtedy, gdy *oid* jest typu $^{\wedge}\tau$ (tzn. *oid* jest wskaźnikiem na wartości typu τ).

Niech jak poprzednio U będzie *typem atomowym*, a $\mathbf{D}$ i $\mathbf{A}$ niech zbiorami odpowiednio stałych atomowych i atrybutów. Niech $\mathbf{ID}$ będzie zbiorem identyfikatorów. Typy wartości obiektowych i typy wskaźnikowe definiujemy następująco.

Definicja 7.3.

1. U jest *typem atomowej wartości obiektowej*, a $^{\wedge}U$ jest *typem wskaźnikowym* na wartości typu U .
2. Jeśli $A_1, \dots, A_n$ są różnymi atrybutami, a $\tau_1, \dots, \tau_n$ są typami wskaźnikowymi lub typami wartości obiektowych, to $\tau, \tau = [A_1 : \tau_1, \dots, A_n : \tau_n]$, jest *typem krotkowej wartości obiektowej*, a $^{\wedge}\tau$ jest *typem wskaźnikowym* na wartości typu τ .
3. Jeśli τ jest typem wskaźnikowym lub typem wartości obiektowej to $\{\tau\}$ jest *typem zbiorowej wartości obiektowej*, a $^{\wedge}\{\tau\}$ jest *typem wskaźnikowym* na wartości typu $\{\tau\}$.

Definicja 7.4.

1. Jeśli $a \in \mathbf{D}$, $i \in \mathbf{ID}$, to (i, a) jest *obiektem atomowym*, tj. typu $(^{\wedge}U, U)$.
2. Jeśli $e_1, \dots, e_n$ są wartościami obiektowymi lub identyfikatorami typów odpowiednio $\tau_1, \dots, \tau_n$, to $(i, [A_1 : e_1, \dots, A_n : e_n])$ jest *obiektem krotkowym* typu $(^{\wedge}[A_1 : \tau_1, \dots, A_n : \tau_n], [A_1 : \tau_1, \dots, A_n : \tau_n])$.

3. Jeśli $e_1, \dots, e_n$ są wartościami obiektowymi lub identyfikatorami tego samego typu τ , to

$(i, \{e_1, \dots, e_n\})$ jest *obiektem zbiorowym* typu $(\wedge\{\tau\}, \{\tau\})$.

Przy takiej definicji typów i obiektów mogą powstać nieskończone pętle. Jaki na przykład byłby typ obiektów $(1, [A : a, B : 2])$ i $(2, [A : b, B : 1])$, które wzajemnie odwołują się do siebie? Bezpośrednie zastosowanie powyższych definicji daje typ nieskończony $[A : U, B : \wedge[A : U, B : \wedge[A : U, B : \wedge[\dots]]]$. Aby uniknąć tego rodzaju anomalii wprowadza się *nazwy typów* i dopuszcza się ich rekurencyjne definiowanie. W rozważanym przypadku możemy wprowadzić typ $T = [A : U, B : \wedge T]$ i wówczas: $(1, [A : a, B : 2]) :: (\wedge T, T)$, $(2, [A : b, B : 1]) :: (\wedge T, T)$.

Przykład 7.3. Poniżej podajemy przykłady obiektów i ich typów (dla uproszczenia piszemy tylko drugą składową typów):

- | | |
|-------------------------------------|---|
| $(1, [A : a, B : 2, C : \{3, 4\}])$ | $:: [A : U, B : \wedge\{U\}, C : \{\wedge U\}]$ |
| $(2, \{c, d\})$ | $:: \{U\}$ |
| $(3, e)$ | $:: U$ |
| $(4, f)$ | $:: U$ |

Patrz również przykład 3.1, gdzie występują obiekty rekurencyjne i współdzielone.

Algorytm 7.2 pokazuje, w jaki sposób przekształcić dowolny obiekt oparty na identyfikatorach za pomocą zbioru obiektów częściowo etykietowanych.

Algorytm 7.2.

Wejście:

Obiekt $o = (i, e)$ i ewentualnie nazwa R (etykieta obiektu o).

Wyjście:

Zbiór obiektów częściowo etykietowanych reprezentujących o i ewentualnie fakt przynależności o do $I(R)$.

Kroki:

Za pomocą operacji pmulti przekształcamy krotki występujące w e na multikrotki. Tak przekształcony obiekt oznaczamy przez (i, c) .

Obiekt (i, c) przekształcamy w zbiór obiektów częściowo etykietowanych:

- jeśli podano nazwę R jako etykietę obiektu o , to stosujemy funkcję *oplo*:
 $oplo(R, (i, c)) = oprep(i, R, c)$,
- jeśli nie podano etykiety obiektu o , to stosujemy funkcję *oplo'*:
 $oplo'((i, c)) = oprep'(i, c)$,

gdzie

```

oprep(i, A, e) =
case e of
    i' : {{(i, A, {i'})}};
    a : {{(i, A, a)}};
    [A1 : e1, ..., An : en] : {{(i, A, {i1 := newId(), ..., in := newId()})}} ∪
        ∪ oprep(i1, A1, e1) ∪ ... ∪ oprep(in, An, en);
    {e1, ..., en} : {{(i, A, {i1 := newId(), ..., in := newId()})}} ∪
        ∪ oprep'(i1, e1) ∪ ... ∪ oprep'(in, en);
endcase

oprep'(i, e) =
case e of
    i' : {{(i, {i'})}};
    a : {{(i, a)}};
    [A1 : e1, ..., An : en] : {{(i, {i1 := newId(), ..., in := newId()})}} ∪
        ∪ orep(i1, A1, e1) ∪ ... ∪ orep(in, An, en);
    {e1, ..., en} : {{(i, {i1 := newId(), ..., in := newId()})}},
        ∪ orep'(i1, e1) ∪ ... ∪ orep'(in, en);
endcase

```

Przykład 7.4.

Zastosowanie algorytm 7.2. do obiektów z przykładu 7.3 daje zbiór

```

1   R {10, 11, 12}
10  A   a
11  B   {2}
2   {13, 14}
13  c
14  d
12  C   {3, 4}
3   e
4   f

```

8. PODSUMOWANIE

W pracy przedstawiono koncepcje bazy danych obiektów częściowo etykietowanych jako bazy danych stanowiącej jednolitą reprezentację danych w procesie integrowania informacji pochodzącej ze źródeł heterogenicznych – zarówno danych sformatowanych, jak i danych semistrukturalnych. Opisano model obiektów częściowo etykietowanych *PLO* (ang. *Partially Labeled Objects*) [14], w skład

którego wchodzi propozycja uniwersum obiektowo-ścieżkowego oraz propozycja języka rachunku ścieżek *PathCal*.

Zbiór obiektów częściowo etykietowanych dobrze nadaje się do jednolitej reprezentacji różnorodnych struktur danych, stąd może być wykorzystany jako model danych dla poziomu osłon w architekturze integrowania danych z wykorzystaniem warstwy osłon i mediatorów (rys. 1). Idea modelu *PLO* wywodzi się z modeli danych opartych na skierowanych grafach etykietowanych i adaptowanych dla potrzeb przetwarzania danych semistrukuralnych i integrowania danych heterogenicznych (wykorzystywanych na przykład w projekcie Tsimmis [19,20,22,27]). W modelu *PLO* zrezygnowano jednak z całkowitego etykietowania dopuszczając jego opcjonalność (częściowość). Ma to istotne konsekwencje dla języka operowania danymi. Problemy te zostały przezwyciężone dzięki wprowadzeniu dwóch ważnych operacji – operacji ? (wydłużenie ścieżek o jeden element) i operacji domknięcia *.

W pracy podano algorytm transformacji różnorodnych struktur danych do zbioru obiektów częściowo etykietowanych. Wykazano między innymi, że istnienie struktur typu „zbiór zbiorów” powoduje, że etykietowanie nie może być całkowite a tylko częściowe.

Problematyka poruszana w pracy należy do ważnego nurtu badań w zakresie przetwarzania danych pochodzących zarówno z tradycyjnych baz danych, jak i z zasobów internetowych. Ma to znaczenie tak dla budowy nowoczesnych systemów informacyjnych, jak i dla systemów eksploracji danych.

9. LITERATURA

- [1] Abiteboul S., *Querying semi-structured data*, Proc. of the International Conference on Database Theory ICDT'97, pp. 1–17.
- [2] Bancilhon F., Khoshafian S., *A calculus for complex objects*, Journal of Computer and System Sciences, No. 38, pp. 326–340, 1989.
- [3] Batini C., Lenzerini M., Navathe S.B., *A Comparative Analysis of Methodologies for Database Schema Integration*, ACM Computing Surveys, Vol. 18, No. 4, 1986, ss. 323–364.
- [4] Fernandez M., Florescu D., Kang J., Levy A., Suciu D., *Catching the boat with Strudel: A Web-site management system*, Proc. ACM-SIGMOD Int. Conf. on the Management of Data, pp. 549–552, 1998.
- [5] Han J. et al., *Dealing with Semantic Heterogeneity by Generalization-Based Data Mining Techniques*, Cooperative Information Systems. Trends and Directions, Academic Press, pp. 207–231, 1998.
- [6] Khoshafian S., *Object-oriented databases*, John Wiley & Sons, New York, 1993.
- [7] Konopnicki D., Shmueli O., *W3QS: A query system for the World Wide Web*, Proc. Int. Conf. on VLDB, 1995.
- [8] Laenens E., Vermeir D., *An overview of OOPS++, an object-oriented database programming language*, In ECOOP'88, Lecture Notes in Computer Science 322, Springer Verlag, pp. 350–373, 1988.
- [9] Lecluse Ch., Richard P., Velez F., *O₂, an object-oriented data model*, Advances in Database Programming Languages, (F. Bancilhon, P. Buneman red.), ACM Press NewYork, Addison-Wesley Publishing Company, Reading, Massachusetts, pp. 257–276, 1990.

- [10] Levy A., Rajaraman A., Ordille J., *Querying heterogeneous information sources using source descriptions*, Proc. 22th Int. Conf. on Very Large Data Bases, pp. 251–262, 1996.
- [11] Lloyd J.W., *Foundations of logic programming*, Second, Extended Edition, Springer-Verlag, Berlin, 1987.
- [12] Ozs M.T., Valduriez P., *Principles of distributed database systems*, Second edition, Prentice-Hall Inc., Upper Saddle River, New Jersey, 1999.
- [13] Pankowski T., *Podstawy baz danych*, Wydawnictwo Naukowe PWN, Warszawa, 1992.
- [14] Pankowski T., *Model bazy danych obiektów częściowo etykietowanych*, Wydawnictwo Politechniki Poznańskiej (w druku).
- [15] Pankowski T., *Podejście obiektowe do integrowania autonomicznych baz danych w system multibazy danych*, Mat. Konferencyjne Informatyka na Wyższych Uczelniach dla Gospodarki Narodowej, Gdańsk, ss. 204–207, 1994.
- [16] Pankowski T., *Przybliżone odpowiedzi na zapytania w kooperacyjnych systemach baz danych*, Mat. Konf. CIR '98, Siedlce, 1998.
- [17] Pankowski T., *Powerdomain of Path for Representing Object Structures*, Fundamenta Informaticae 33(2), ss. 121–148, 1998.
- [18] Pankowski T., *Semantics of approximate answers in cooperative database systems*, Proc. of Int. Conf. on Computational Intelligence on Modeling, Control and Automaton, CIMCA '99, Vienna, February 17–19, 1999.
- [19] Papakonstantinou Y., Garcia-Molina H., Widom J., *Object exchange across heterogenous information sources*, Proc. of 11th Conference on Data Engineering, Taipei, Taiwan, pp. 252–260, 1995.
- [20] Papakonstantinou Y., Abiteboul S., Garcia-Molina H., *Object Fusion in Mediator Systems*, Proc. of the 22nd VLDB Conference, 1996.
- [21] Piatesky-Shapiro G., *Data mining and knowledge discovery in business databases*, Foundations of Intelligent Systems, LNIAI 1079, Springer-Verlag Berlin, pp. 56–67, 1996.
- [22] Quass D., Rajaraman A., Sagiv Y., Ullman J., Widom J., *Querying Semistructured Heterogenous Information*, Proc. of International Conference on Deductive and Object-Oriented Databases, DOOD95, LNICS 1013, pp. 319–344, 1995.
- [23] Reiter R., *Towards a logical reconstruction of relational database theory*, On Conceptual Modelling (reds.: M. Brodie, J. Mylopoulos, J.W. Schmidt), Springer-Verlag Berlin, pp. 191–233, 1984.
- [24] Subieta K., Leszczyłowski J., Ullidowski I., *Processing Semi-Structured Data in Object Bases*, Prace IPI PAN 852, Instytut Podstaw Informatyki Polskiej Akademii Nauk, Warszawa, 1998.
- [25] Tomasic A., Amouroux R., Bonnet P., Kapitskaia O., Naacke H., Raschid L., *The Distributed Information Search Component (DISCO) and the World-Wide Web – Prototype demonstration*, Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 546–548, 1997.
- [26] Ullman J.D., *Principles of Database and Knowledge – Base Systems*, Vol. I and Vol. II, Computer Science Press, 1988/1989.
- [27] Ullman J.D., *Information integration using logical views*, [in:] Proc. of the International Conference on Database Theory ICDT '97, 1997.

PARTIALLY TAGGED OBJECT DATABASE MODEL IN THE PROCESS OF INTEGRATING DATA

A model of schemaless database based on idea of tagged objects is presented. The model can be used to creating databases in systems that integrate data from different sources (i.e. HTML, XML, SGML, LaTeX formats).

CZĘŚĆ II

ROZWIĄZANIA
PRAKTYCZNE

Artur WILCZEK*

WYBRANE WŁASNOŚCI OBIEKTOWE RELACYJNYCH BAZ DANYCH

Zastosowania baz danych wymagające modelowania danych o dużej złożoności strukturalnej i nietrywialnych własnościach behawioralnych wymagają stosowania nowych bardziej złożonych modeli danych. Modele te powstają jako rozszerzenie modelu relacyjnego o własności obiektowych języków programowania lub jako rozszerzenie języków obiektowych o funkcjonalność systemów baz danych. Praca przedstawia wybrane własności obiektowe języka SQL wprowadzone w SQL3 oraz wzorowane na tych propozycjach rozszerzenia obiektowe Oracle8.

1. WPROWADZENIE

Podejście obiektowe do konstruowania systemów informatycznych zdobyło sobie liczne grono zwolenników. Obiektowość jest coraz częściej obecna w produktach przemysłu informatycznego. Trendy te nie ominęły także baz danych. Powstają zarówno laboratoryjne jak i komercyjne obiektowe systemy zarządzania bazami danych.

Producenci i dostawcy relacyjnych baz danych takich jak **DB2**, **Informix**, **Ingres**, **Sybase** czy **Oracle** znaleźli się pod naciskiem dwóch tendencji. Z jednej strony rynek wymógł wzbogacenie technologii relacyjnej o takie cechy jak procedury bazy danych, aktywne reguły (triggery), języki 4GL (np. Oracle Forms) będące hybrydami technik programowania obiektowego, języka zapytań i uniwersalnego języka programowania. Z drugiej strony popularność obiektowości spowodowała pokusę wprowadzenia wielu cech obiektowych, takich jak klasy, metody, abstrakcyjne typy danych, pozwalające na głoszenie twierdzeń o „częściowej obiektowości” systemów relacyjnych. Tego rodzaju rozszerzenia wprowadzono do większości relacyjnych baz danych znajdujących się na rynku. Reprezentantem tego podejścia jest standard SQL3, opracowany przez komitet standaryzacyjny ISO, będący następcą i rozszerzeniem SQL-92 oraz model danych systemu Oracle8. Praca przedstawia wybrane własności obiektowe języka SQL wprowadzone w SQL3 oraz wzorowane na tych propozycjach rozszerzenia obiektowe Oracle8.

* Wydziałowy Zakład Informatyki, Wydział Informatyki i Zarządzania Politechniki Wrocławskiej, 50-370 Wrocław, Skwer Idaszewskiego 9, wilczek@ci.pwr.wroc.pl.

2. WŁASNOŚCI OBIEKTOWE SQL3

2.1. TYPY DANYCH

W relacyjnym modelu danych zbiór wartości atrybutów ograniczony jest do wartości atomowych. Definicje atrybutów relacji mogą odwoływać się do ograniczonego zbioru predefiniowanych prostych typów danych, takich jak: character, number, date. Własność ta stanowi istotne utrudnienie modelowania struktur danych.

Standard języka SQL3 umożliwia definiowanie przez użytkownika własnych „zrozumiałych” dla SZBD złożonych typów danych, które mogą być wykorzystane przy definicji atrybutów relacji.

Ilustruje to poniższy przykład, w którym zdefiniowano złożony typ danych, następnie wykorzystany jako typ atrybutu relacji.

```
CREATE TYPE adres (
    miasto char(30),
    ulica   char(30),
    nr      integer,
    kod     char(6));
```

```
CREATE TABLE osoby(
    imie          char(30),
    nazwisko      char(30),
    adres_staly   adres,
    adres         adres);
```

Odwołanie do wartości składowych atrybutów złożonych następuje poprzez wykorzystanie operatora ‘..’. Przykładem zdania wykorzystującego ten operator jest poniższe zdanie, w którym warunkiem zapytania jest wartość składowa adres..ulica atrybutu złożonego adres, a wynikami zapytania wartości składowe adres..miasto, adres..kod.

```
SELECT nazwisko,adres..miasto,adres..kod
FROM osoby
WHERE adres..ulica = "WITTIGA";
```

Znaczącym rozszerzeniem przewidzianym w SQL3 jest możliwość definiowania abstrakcyjnych typów danych (ADT). Abstrakcyjne typy danych definiują równolegle atrybuty i operacje.

W poniższym przykładzie zdefiniowano ADT pracownik, który jest zbiorem atrybutów oraz funkcji na nich operujących.

```
CREATE TYPE pracownik(
    PUBLIC
        imie            char(30),
        nazwisko        char(30)
        FUNCTION pensja(oA osoba) RETURNS number;
    BEGIN
        RETURN oA.stawka;
    END;
    FUNCTION awans(oA osoba, kwota number)
    RETURNS osoba
    BEGIN
        set oA.stawka = oA.stawka + kwota;
        RETURN oA
    END;
PRIVATE
    stawka number; );

CREATE TABLE pracownicy(
    dane pracownik);

SELECT pensja(dane) FROM pracownicy
WHERE dane..nazwisko = 'NOWAK';
```

Operatory zasięgu pozwalają na ukrycie przed użytkownikiem wystąpień typu od jego implementacji. W przykładzie powyższym ukryto atrybut stawka. Dostęp do informacji o pensji osoby może zostać zrealizowany jedynie przez związane z typem operacje awans, pensja.

2.2. DZIEDZICZENIE

Definicja nowego typu danych (podtypu) może bazować na innym uprzednio zdefiniowanym typie. Podtyp dziedziczy wszystkie własności nadtypu (atrybuty i operacje), dodając własne atrybuty i operacje. W poniższym przykładzie zdefiniowano nowy typ danych kierownik w oparciu o typ pracownik oznacza to, że typ kierownik obejmuje atrybuty odziedziczone oraz zdefiniowane lokalnie: jednostka, podwładni, nowy typ nie wnosi nowych funkcji ponad dziedziczone. Atrybut podwładni jest atrybutem wielowartościowym, zdefiniowanym w oparciu o jeden z dostępnych

konstruktorów kolekcji. Kolekcje są to wartości zawierające pewną liczbę jednorodnych elementów.

SQL3 przewiduje następujące kolekcje:

- zbiory (SET) - elementy bez powtórzeń, kolejność nieistotna
- listy (LIST) - elementy z powtórzeniami, kolejność istotna
- wielozbiór (MULTISET) - elementy z powtórzeniami, kolejność nieistotna

W poniższym przykładzie najpierw tworzony jest typ danych pracownicy jako kolekcja wystąpień typu pracownik, następnie typ kierownik dziedziczący po typie pracownik z własnym atrybutem wielowartościowym podwładni.

```
CREATE TYPE pracownicy (SET(pracownik));
CREATE TYPE kierownik UNDER pracownik (
PUBLIC
    jednostka char(30),
    podwladni pracownicy );
```

W zapytaniach wystąpienia kolekcji (typu kontenerowego) są traktowane jak tablice.

```
SELECT jednostka FROM kierownik k
WHERE 'NOWAK' IN
(SELECT p..nazwisko FROM TABLE (k.podwladni) p);
```

2.3. TYP KROTKOWY

Aby umożliwić nadanie własności obiektowych całym wierszom relacji wprowadzono do SQL3 typ krotkowy (ang. named row type). Wszystkie krotki relacji zdefiniowanej w oparciu o typ krotkowy będą obiektami. W momencie ich tworzenia nadać im można unikalny systemowy identyfikator, co obrazuje poniższy przykład.

```
CREATE TYPE adres (
    id_adresu      REF (adres),
    miasto char(30),
    ulica          char(30),
    nr             integer,
    kod            char(6));

CREATE TABLE adresy OF TYPE adres
(VALUE FOR id_adresu ARE SYSTEM GENERATED);
```

2.4. TYPY REFERENCYJNE

Nadanie krotkom unikalnych identyfikatorów pozwoliło wprowadzić do modelu danych jawnego modelowania związków łączących krotki relacji. Związki te są definiowane przez odwołanie się do typu referencyjnego, którego wystąpieniami są unikalne identyfikatory krotek. Identyfikatory krotek są trwałe i niezmiennie dla całego życia danego wystąpienia.

Wartość typu referencyjnego może być zapamiętana w tablicy i następnie użyta jako bezpośrednia referencja (wskaznik) do specyficznego wiersza w innej tablicy. Pozwala to na zagnieżdżanie krotek jako wartości atrybutów innych krotek. W poniższym przykładzie wartościami atrybutu adres są krotki relacji adresy.

```
CREATE ROW TYPE pracownik(
    id_prac REF(pracownik),
    imie      char(30),
    nazwisko  char(30),
    adres     REF(adres));

CREATE TABLE pracownicy OF TYPE pracownik
(SCOPE FOR adres IS adresy,
VALUES FOR id_prac ARE SYSTEM GENERATED);
```

Jawne odwołanie do identyfikatora krotki możliwe jest za pomocą operatora REF ROW. Odwołanie do wartości atrybutów lub metod zagnieżdżonej krotki jest realizowane przez operator dereferencji '->'.

```
INSERT INTO pracownicy (nazwisko, szef)
VALUES('NOWAK',(SELECT REF ROW p
                  FROM pracownicy p
                  WHERE nazwisko = 'KOWALSKI')));

SELECT adres->miasto FROM pracownicy
WHERE nazwisko = 'NOWAK';
```

3. WŁASNOŚCI OBIEKTOWE ORACLE8

Wersja 8 systemu zarządzania bazami danych Oracle została wzbogacona o elementy obiektowe wzorowane na propozycjach standardu SQL3. Należy pamiętać, że

nie są one integralną częścią serwera – można z nich korzystać po zainstalowaniu rozszerzenia obiektowego (ang. *Object Option*). Poniżej przedstawiona zostanie podstawowa własność obiektowa Oracle8: możliwość definiowania typów danych.

3.1. OBIEKTOWE TYP DANYCH

Użytkownik może definiować w Oracle8 typy obiektowe, które później mogą służyć do tworzenia ich trwałych bądź ulotnych wystąpień. Trwałe obiekty będą pamiętane w bazie danych, natomiast obiekty ulotne powstają jako zmienne bloków PL/SQL.

Definicja typu obiektowego składa się z jego specyfikacji oraz implementacji. Specyfikacja typu to interfejs zawierający definicje atrybutów oraz deklarację metod. Implementacja typu obejmuje implementację metod zadeklarowanych w specyfikacji.

```
CREATE TYPE pracownik AS OBJECT(
    imie          char(30),
    nazwisko      char(30),
    stawka        number; );
```

Wykonanie powyższego zdania spowoduje utworzenie specyfikacji klasy, która jest dostępna do jej jawnego usunięcia poleceniem DROP TYPE. Typ obiektowy pracownik nie posiada metod, dlatego też nie ma potrzeby specyfikowania jego implementacji.

Oracle8 nie wymusza pełnej hermetyzacji obiektów. Do atrybutów obiektu odwołać się można bezpośrednio zarówno z poziomu języka PL/SQL jak i SQL. Nic jednak nie stoi na przeszkodzie by dostęp do pewnych danych obudować interfejsem proceduralnym, co obrazuje przykład:

```
ALTER TYPE pracownik REPLACE AS OBJECT(
    imie          char(30),
    nazwisko      char(30),
    stawka        number;
MEMBER FUNCTION pensja RETURN number;
MEMBER PROCEDURE awans(kwota number)
PRAGMA RESTRICT_REFERENCES (pensja,WNDS,RNPS,WNPS)
PRAGMA RESTRICT_REFERENCES (awans,RNPS,WNPS));
```

Klauzula PRAGMA jest wykorzystywana do określenia, czy dana metoda może być wywołana bez narażania się na niekorzystne objawy uboczne. W zapytaniach nie można stosować metod zmieniających stan bazy danych, gdyż wynik takiego zapytania mógłby być niedeterministyczny. Podobny mechanizm stosowany jest w przypadku funkcji i procedur składowanych Oracle7.

W powyższym przykładzie funkcja pensja nie modyfikuje danych w bazie danych, może więc być wykorzystywana w zdaniach select, insert, delete i update SQL. Natomiast procedurę awans można wykorzystywać w innych metodach lub programach PL/SQL.

Zadeklarowane w specyfikacji typu metody muszą zostać zaimplementowane, tak jak w poniższym przykładzie.

```
CREATE OR REPLACE TYPE BODY pracownik AS(
    MEMBER FUNCTION pensja RETURNS number IS
    BEGIN
        RETURN stawka;
    END;
    MEMBER PROCEDURE awans (kwota number) IS
BEGIN
    stawka := stawka + kwota;
END;
END;
```

Poniższy przykład pokazuje powołanie do życia ulotnego obiektu typu pracownik.

```
DECLARE
    prac pracownik;
BEGIN
    prac:=pracownik('JAN','NOWAK',1200);
    prac.imie := 'JAN, PAWEL';
    prac.awans(200);
END;
```

Wywołanie metody systemowej pracownik()- konstruktora, tworzy obiekt i inicjuje atrybuty tego obiektu wartościami argumentów wywołania konstruktora. Konstruktor ma taką samą nazwę jak typ, którego wystąpienia tworzy.

Utworzony w powyższym przykładzie obiekt ma charakter lokalny i po zakończeniu działania programu PL/SQL przestaje istnieć.

Aby utrwalić obiekty w bazie danych należy utworzyć relację, której atrybuty lub cały wiersz zostaną zdefiniowane w oparciu o typ obiektowy. Odpowiednio obiekty takie nazywane są obiektami atrybutowymi bądź wierszowymi. Poniższe przykłady obrazują ich wykorzystanie.


```
CREATE TABLE tab1 (
    prac pracownik;
    id      number);
```

```
INSERT INTO tab1
VALUES(pracownik('JAN','NOWAK',1200),1));
```

Wartość wstawionego wiersza można wyświetlić zdaniem:

```
SELECT prac,id,prac.imie FROM tab1;
```

co da następujący wynik:

PRAC(IMIE,NAZWISKO,STAWKA)	ID	IMIE
----- -- ----		
PRACOWNIK('JAN','NOWAK',1200)	1	JAN

Krotki relacji tab2 to obiekty krotkowe.

```
CREATE TABLE tab2 OF pracownik;
INSERT INTO tab2 VALUES(PACOWNIK('JAN','NOWAK',1200));
```

Do wartości wstawionego obiektu można odwołać się za pomocą operatora VALUE.
Wykonanie zdania:

```
SELECT VALUE(t) FROM tab2 t;
```

zwróci w wyniku:

PRAC(IMIE,NAZWISKO,STAWKA)

PRACOWNIK('JAN','NOWAK',1200)

natomiast wykonanie zdania:

```
SELECT * FROM tab2;
```

spowoduje wyświetlenie:

IMIE	NAZWISKO	STAWKA
----	-----	-----
JAN	NOWAK	1200

W zdaniach SQL można także wykorzystywać metody o odpowiedniej „pragmie”. Przykładem wykorzystania metody w zdaniu SQL jest zdanie:

```
SELECT p.pensja FROM tab2;
```

którego wykonanie zwróci w naszym przykładzie:

P.PENSJA()

1200

3.2. TYP REFERENCYJNY

Do specyfikacji związków między obiektami służy typ referencyjny. Przy wykorzystaniu typu referencyjnego można odczytywać zarówno wartości atrybutów zagnieżdżonych obiektów jak i unikalne identyfikatory obiektów. Tożsamość (identyfikator) posiadają tylko obiekty wierszowe składowane w tabelach obiektowych

```
CREATE TYPE pracownik AS OBJECT(
    imie          char(30),
    nazwisko      char(30),
    szef  REF pracownik; );
```

```
CREATE TABLE pracownicy OF pracownik;
```

Wykonanie operacji dereferencji w sposób niejawny umożliwia pobranie wartości atrybutu obiektu

```
SELECT p.szef.nazwisko FROM pracownicy p
WHERE nazwisko = 'NOWAK';
```

Dereferencja jawna:

```
SELECT deref(p.szef) FROM pracownicy p
WHERE nazwisko = 'NOWAK';
```

umożliwia pobranie całego obiektu zagnieżdżonego.

Utworzenie obiektów i ich powiązanie obrazuje kolejny przykład:

```

DECLARE
    s REF pracownik;
BEGIN
    INSERT INTO pracownicy p
    VALUES(PRACOWNIK('PIOTR','KOWALSKI',null))
    RETURNING ref(p) INTO s;
    VALUES(PRACOWNIK('PIOTR','KOWALSKI',s));
    COMMIT;
END;
```

4. PODSUMOWANIE

Bazy danych pozostają ciągle dynamicznie rozwijająca się dziedziną. Wobec nowych wyzwań i zastosowań cechowanych dużą złożonością informacyjną poszukuje się nowego standardu języka baz danych. Interesująca propozycją jest standard SQL3. Własności obiektowe w bazach danych są ciągle otwartym i intensywnie rozwijanym obszarem badań. Z tego powodu, wypracowane w SQL3 propozycje przypuszczalnie podlegać będą dalszej ewolucji.

LITERATURA

- [1] American National Standards Institute (ANSI) Database Committee (X3H2). Database Language SQL3, J. Melton (ed.), 1994.
- [2] Melton J.. *Object Technology and SQL*, Adding Objects to a Relational Language. Bulletin On Data Engineering (IEEE), 1994.

OBJECT-ORIENTED TECHNIQUES IN RELATIONAL DATABASES

Modern databases practical applications with complex data and behaviour modelling uses new data model which arose as object-oriented functional extension of relational data model or database functional extension of object-oriented programming. This paper gives a brief overview of object-oriented properties introduced in SQL3 standard and their implementation in Oracle8.

*Borland Visual dBASE 7.0,
Explorator Projektów,
Source Editor,
SQL Designer*

Hanna MAZUR*
Krzysztof MICHALAK**

BORLAND VISUAL DBASE 7.0 – NOWOCZESNE ŚRODOWISKO DO TWORZENIA APLIKACJI BAZODANOWYCH

Artykuł omawia nowe elementy systemu Borland Visual dBASE 7.0 w stosunku do poprzednich wersji systemów dBASE. Zwraca uwagę na zalety i wady nowego systemu. Borland Visual dBASE 7.0 umożliwia zaawansowane programowanie zorientowane obiektowo przy czym dane są przechowywane w tabelach (zgodnie z modelem relacyjnym).

1. WSTĘP

Visual dBASE 7 firmy Borland jest 32-bitowym środowiskiem do tworzenia aplikacji bazodanowych pod WINDOWS 95 lub WINDOWS NT. Środowisko to oferuje interaktywne narzędzia do zarządzania bazą danych, daje możliwość wizualnego konstruowania aplikacji oraz posiada wbudowany język oparty na modelu obiektowym. Ponadto, środowisko Visual dBASE 7 zawiera elementy znane z wcześniejszych wersji produktu, z którymi jest w znacznym stopniu kompatybilne. Dzięki zastosowaniu zespołu bibliotek **BDE (Borland Database Engine)** istnieje możliwość tworzenia i obsługi tabel w formatach dBASE, Paradox, FoxPro, MS Access a także dowolnego innego systemu zarządzającego bazą danych, o ile ten obsługuje mechanizm **ODBC (Open Database Connectivity)**. Dodatkowe sterowniki umożliwiają połączenie aplikacji stworzonej w środowisku dBASE z bazami danych stworzonymi za pomocą produktów Oracle, Sybase, InterBase, MS SQL Server, IBM DB2 i Informix.

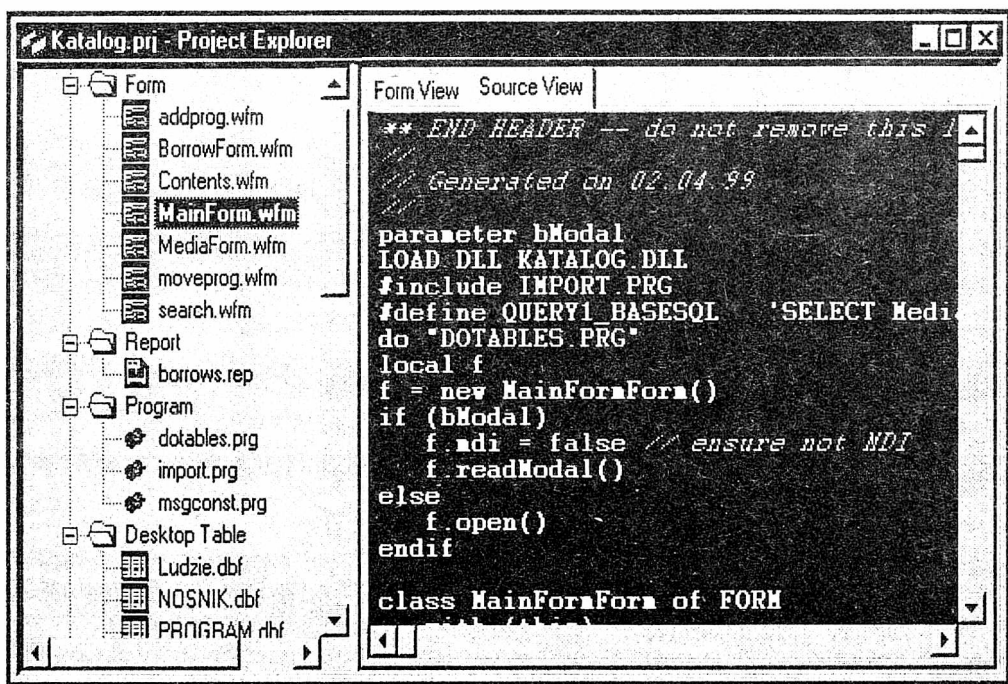
* Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Wydziałowy Zakład Informatyki.

** Politechnika Wrocławska, Wydział Informatyki i Zarządzania – student IV roku Inżynierii Oprogramowania.

2. CO NOWEGO W WERSJI 7.0?

Wersja 7.0 środowiska dBASE zawiera szereg ulepszeń i rozszerzeń w porównaniu z poprzednimi wersjami, zarówno jeśli chodzi o dostępne w środowisku „wyposażenie”, jak i w zakresie języka programowania.

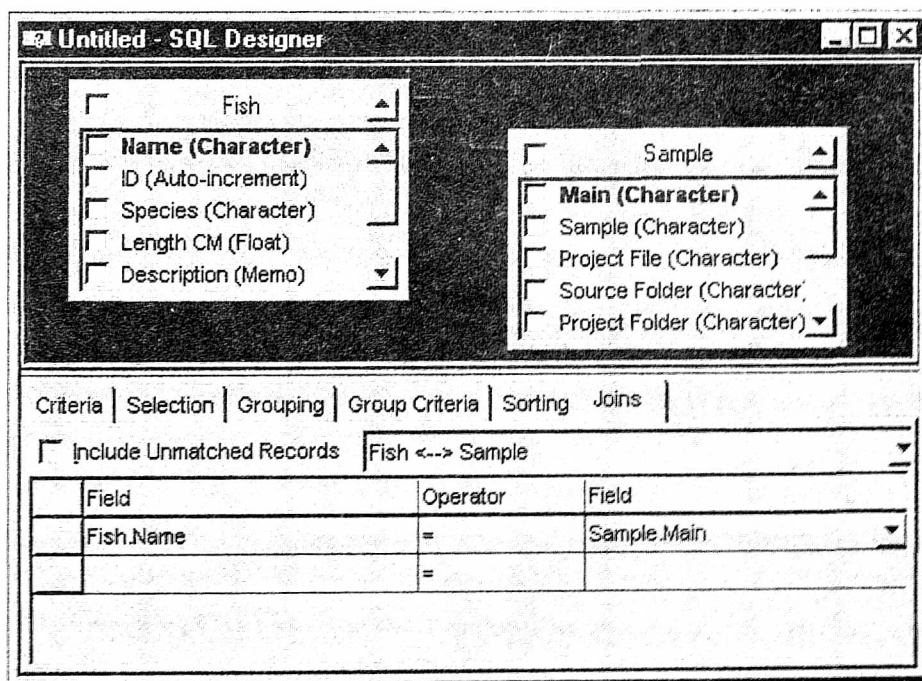
Wprowadzono zintegrowany moduł do tworzenia raportów obsługujący cały szereg klas związanych z tworzeniem raportów: **Report**, **Group**, **PageTemplate**, **StreamSource**. Zaletą systemu raportów w dBASE 7 jest jego duża elastyczność i możliwość łatwego różnicowania wyglądu poszczególnych stron raportu zależnie od potrzeb. Oprócz typowych zestawień można, na podstawie danych zawartych w tabelach, tworzyć etykiety z adresami do korespondencji seryjnej. Proces tworzenia raportów i etykiet jest – podobnie jak proces projektowania aplikacji – oparty na wizualnym modelu pracy. Środowisko dBASE 7 oferuje możliwość posługiwania się kreatorami automatyzującymi proces tworzenia raportów (Crystal Reports). Pakiet VdB7 zawiera usługi konwertujące raporty z poprzednich wersji dBASE do nowego formatu.



Rys. 1. Eksplorator projektów
Fig. 1. Project Explorer

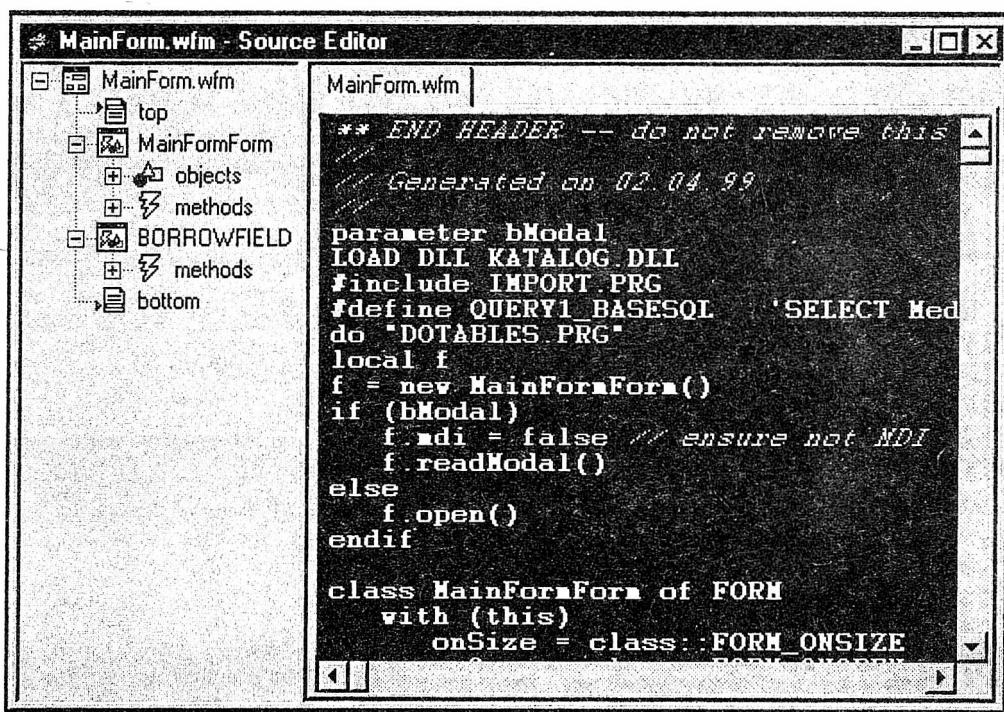
Następca systemu Katalogów – **Eksplorator Projektów** (Project Explorer), daje możliwość łatwego zarządzania wszystkimi plikami wchodzącymi w skład tworzonej aplikacji. **Eksplorator Projektów** umożliwia przeglądanie wszystkich plików wchodzących w skład projektu w postaci tekstowej, kodu źródłowego oraz, w przypadku formularzy i raportów, w postaci wizualnego podglądu tworzonego widoku. Istnieje możliwość kompilowania i uruchamiania (także za pomocą debuggera) całej aplikacji lub pojedynczych widoków z poziomu **Eksploratora Projektów**.

Klasy zapewniające dostęp do tabel wykorzystują mechanizm zapytań SQL. Daje się tu zauważyć różnicę pomiędzy systemem dBASE 7 a takimi środowiskami do tworzenia aplikacji bazodanowych jak np. C++Builder czy Delphi. W systemie dBASE nie występują obiekty bezpośrednio odwzorowujące tabele (takie jak np. klasa **TTable** w C++Builderze), zamiast tego dostęp do danych zawartych w tabelach zapewniają obiekty klasy **Query**, natomiast obsługa plików (w tym np. tworzenie indeksów) jest przeniesiona do obiektów klasy **Database**. Osoby przyzwyczajone do Delphi czy CBuildera, gdzie całą obsługę tabel można zamknąć w obiekcie klasy **TTable**, mogą takie rozdzielenie funkcjonalności na obiekty dwóch różnych klas odczuć jako znaczny dyskomfort. Zaletą używania wyłącznie obiektów typu **Query** jest natomiast duża elastyczność zapytań SQL – dzięki temu nie ma problemów nawet przy znacznej modyfikacji źródła danych.



Rys. 2. SQL Designer
Fig. 2. SQL Designer

Korzystanie z zapytań SQL w systemie Borland Visual dBASE 7 jest znacznie prostsze niż w poprzednich wersjach. Komenda **SQLExec** umożliwia wykonywanie poleceń tego języka, odwołujących się bezpośrednio do danych zapamiętanych w plikach *.DBF. Zrezygnowano z nienaturalnego, we wcześniejszych wersjach dBASE, wymogu przeddefiniowywania plików *.DBF na tabele SQL. Polecenia **DBDEFINE** i **CREATE DATABASE** nie są już potrzebne i zniknęły z zasobu komend języka. Specjalne narzędzie do tworzenia zapytań SQL upraszcza opracowywanie zapytań złożonych. **SQL Designer** daje możliwość wizualnego projektowania połączeń między tabelami, grupowania rekordów oraz nakładania ograniczeń na generowany przez zapytanie zbiór rekordów. **SQL Designer** zastąpił dotychczas używany system zapytań **QBE (Query By Example)**, który nie jest dostępny w dBASE 7.



Rys. 3. Edytor kodu źródłowego
Fig. 3. Source Editor

Wprowadzono nowe typy pól danych: **Double**, **Long**, **TimeStamp** oraz **AutoIncrement**. Zwłaszcza pola tego ostatniego typu mogą być bardzo przydatne, gdyż system automatycznie dba o to, aby wartości tych pól były unikalne. Efektem rozwiązania przyjętego w dBASE 7 jest fakt, że wartość pola typu **AutoIncrement** dla nowo tworzonego rekordów jest zawsze zwiększana o jeden i nie ma możliwości

wprowadzenia zmian (także luki po usuniętych rekordach nie są wypełniane). System dBASE 7 pozwala na tworzenie indeksów typu **DISTINCT** uniemożliwiających wprowadzenie do tabeli dwóch rekordów o tej samej wartości danego pola. Niestety, w dBASE 7 pozostały niektóre ograniczenia występujące w poprzednich wersjach (np. maksymalna długość wyrażenia indeksowego wynosi 100 znaków).

Środowisko dBASE 7 zawiera ulepszony edytor kodu źródłowego (Source Editor) podświetlający składnię programu oraz wyposażony w podgląd zawartości modułu w postaci drzewa zawierającego wszystkie klasy, obiekty i metody zdefiniowane w module.

Znacznie rozszerzono język programowania systemu. W Borland Visual dBASE 7 wprowadzono szereg nowych klas, w tym widoki (**Grid**), paski postępu (**Progress indicator**), obiekty wyświetlające drzewo (**TreeView**), paski narzędzi (**Toolbar**) oraz obiekty obsługujące pliki (**File**). Definicję języka poszerzono o liczne elementy zaczerpnięte z języków takich jak Pascal czy C++. Do najważniejszych należą:

- operatory:
 - % modulo
 - ++ inkrementacja
 - dekrementacja
 arytmetyczne operatory przypisania:
 - +=
 - =
 - *=
 - /=
 - %=
- możliwość zaznaczania komentarzy symbolami znanymi z C++:
 - /* Tekst komentarza */
 - oraz
 - // Komentarz do końca linii.
- stałe i operatory logiczne zgodne z powszechnie przyjętym standardem: **false, true, and, or, not**
- operator przypisania **:=** (*ang. assignment-only operator*) różny od operatora **=** (*ang. assign/create operator*) i zapewniający (w przeciwieństwie do tego ostatniego), że przypisanie będzie dokonywane tylko do wcześniej zadeklarowanej zmiennej (jeśli odpowiednia zmienna nie została wcześniej zadeklarowana, zostanie to potraktowane jako błąd składni). Stosowanie operatora **:=** uodparnia w znacznym stopniu kod na tzw. „literówki”

- obsługa wyjątków z możliwością przechwytywania błędów występujących w predefiniowanych funkcjach i metodach a także z możliwością tworzenia i wywoływania wyjątków zdefiniowanych przez użytkownika
- konstrukcja **WITH** skracająca zapis przy dostępie do wielu pól lub metod tego samego obiektu.

Znany z wcześniejszych wersji dBASE mechanizm transakcji może być w dBASE 7 obsługiwany za pomocą funkcji analogicznych jak we wcześniejszych wersjach oraz za pomocą metod klasy **Database**.

System Visual dBASE 7 umożliwia zabezpieczanie tabel za pomocą haseł oraz praw dostępu. Zabezpieczane tabele są kodowane przez system. Istnieje możliwość nadawania użytkownikom uprawnień do całych tabel bądź do poszczególnych rekordów.

3. ŚRODOWISKO DBASE 7 – ZA I PRZECIW

Do zalet nowego środowiska należą:

- Zastosowanie zespołu bibliotek BDE (Borland Database Engine). Dzięki ogromnej elastyczności systemu BDE aplikacje napisane w dBASE 7 mogą obsługiwać tabele w wielu popularnych formatach. Istnieje także możliwość połączenia aplikacji z „dużymi” systemami zarządzania bazami danych takimi jak np. Oracle czy InterBase.
- Tworzenie aplikacji w sposób wizualny, zwalniający programistę z konieczności żmudnego projektowania interfejsu oraz dający możliwość wygodnego posługiwania się obiektami. Znaczne podobieństwa do takich narzędzi jak C++Builder czy Delphi ułatwiają pracę z programem.
- Dająca się zauważyć ewolucja języka dBASE w kierunku powszechnie znanych języków programowania (C++, Pascal) ułatwia nowym użytkownikom tworzenie aplikacji. Osoby zapoznane z wcześniejszymi wersjami systemu dBASE mogą używać starej składni poleceń, która nadal jest rozpoznawana przez system chociaż w wielu miejscach uległa drobnym zmianom mającym na celu dostosowanie języka do obiektowego modelu programowania.
- System dBASE 7 obsługuje kilka użytecznych standardów wymiany danych przyjętych w środowisku Windows. Zarówno aplikacje napisane w dBASE 7 jak i tabele w formacie DBF7 mogą obsługiwać obiekty **OLE**.

Klasy **DDELink** i **DDETopic** umożliwiają nawiązywanie transakcji **DDE**. Uruchamianie innych programów bezpośrednio z wnętrza aplikacji jest możliwe poprzez komendę **RUN**, a wywoływanie dowolnych funkcji zawartych w 32-bitowych bibliotekach **DLL** za pomocą polecenia **LOAD DLL** oraz deklaracji z użyciem słowa kluczowego **EXTERN**. Zwłaszcza ta ostatnia możliwość znacznie rozszerza funkcjonalność systemu.

- Możliwość wygenerowania w pełni funkcjonalnego zbioru *.EXE zawierającego aplikację. Do działania aplikacji niezbędne są: biblioteka uruchomieniowa (run-time) oraz zespół bibliotek BDE.

Niestety twórcy systemu dBASE 7 nie ustrzegli się pewnych usterek i niedociągnięć utrudniających pracę z systemem i stworzonymi w nim aplikacjami.

- Zdecydowanym minusem systemu jest fakt, że kompilator języka dBASE 7 nie generuje kodu maszynowego procesora a jedynie półskompilowany pseudokod poddawany podczas wykonania dalszej interpretacji. Spowalnia to znacznie działanie programu - w stosunku do programów napisanych w językach w pełni kompilowalnych spadek prędkości jest dotkliwy nawet na szybkich komputerach.
- Irytującą cechą wizualnego modułu do projektowania widoków jest fakt, że deklaracje obiektów są w trakcie projektowania tworzone całkowicie od zera a nie jedynie modyfikowane co powoduje, że modyfikacje dokonane przez programistę w kodzie w części deklaracyjnej obiektu są traczone.
- Kolejna grupa usterek zdaje się mieć swoje źródło w błędnym, w niektórych sytuacjach, działaniu menedżera pamięci. Niektóre obiekty nie są prawidłowo zwalniane podczas zamykania aplikacji (dotyczy to zwłaszcza obiektów klas **Database** i **Session**) i jeśli nie zwolni ich się explicite mogą powodować błędy przy opuszczaniu aplikacji bądź całego środowiska. Także wywołanie funkcji **SQLExec** może pociągać za sobą błąd objawiający się niekontrolowaną zmianą wartości wskaźnika **THIS**.
- Cechą, którą trudno uznać za błąd, ale która może utrudnić pracę osobom przyzwyczajonym do poprzednich wersji systemu, jest brak systemu zapytań QBE. Został on zastąpiony systemem tworzenia zapytań SQL zwanym **SQL Designer**. Ten ostatni, jakkolwiek bardzo wygodny i nie wymagający znajomości języka SQL(!), pod względem obsługi odbiega znacznie od QBE i wymaga od użytkownika wyrobienia nowych nawyków.
- Poważnym niedociągnięciem (przynajmniej w wersji angielskiej produktu) jest błędna obsługa liter spoza alfabetu angielskiego, w szczególności polskich znaków diakrytycznych. Błędna praca systemu w tym zakresie może powodować, że polskie litery nie będą prawidłowo wyświetlane na ekranie lub nawet nie zostaną prawidłowo zapisane do tabeli!

4. PODSUMOWANIE

Po zapoznaniu się z systemem Visual dBASE 7 widać, że system znacznie zmienił się na korzyść w stosunku do poprzednich wersji. Dzięki zaimplementowaniu wielu mechanizmów systemu Windows, wizualnemu modelowi pracy z programem oraz ewolucji języka programowania w kierunku obiektowego modelu programowania, system Visual dBASE 7 stał się systemem umożliwiającym tworzenie funkcjonalnych, i co nie mniej ważne, wygodnych i zgodnych z przyzwyczajeniami użytkownika aplikacji. Wspecjalizowanie środowiska dBASE w zakresie tworzenia aplikacji bazodanowych znacznie przyspiesza i ułatwia tworzenie takich aplikacji w porównaniu z językami programowania ogólnego przeznaczenia. Jedynym poważnym minusem systemu dBASE jest stosunkowo duża liczba błędów występujących podczas działania aplikacji. Zmusza to do dłuższego i bardziej szczegółowego testowania aplikacji dBASE niż tych tworzonych np. w Delphi. Narzędzie dBASE wydaje się być więc optymalne do tworzenia małych i średnich aplikacji. Tworzenie dużych systemów w Visual dBASE 7.0 może okazać się po prostu zbyt pracochłonne w porównaniu z ich tworzeniem za pomocą innych narzędzi.

LITERATURA

- [1] *Borland Visual dBASE for Windows 95 and Windows NT, version 7. Developer's Guide.* USA, 1997.
- [2] Hoffbauer M., Feil P., *dBASE 5 dla Windows, wersja polska.* MIKOM, Warszawa, 1995.
- [3] Jasiński P.J., Królikowski Z., Szulczyński J., *Mikrokomputerowe Bazy danych z programowaniem w dBASE IV.* Nakom, Poznań, 1992.
- [4] Mazur H., Mazur Z., *Programowanie w dBASE IV.* PSZI. Wrocław, 1997.
- [5] Simpson A., Rinehart M., *Zrozumieć dBASE 5 for Windows*, t. I, II. Wydawnictwo PLJ, Warszawa, 1996.
- [6] Strużińska-Walczak A., Walczak K., *Borland dBASE IV, wersja 2.0. Programowanie. SQL.* Wydawnictwo W&W, Warszawa, 1993.
- [7] Strużińska-Walczak A., Walczak K., *dBASE IV. Programowanie. Help*, Warszawa, 1990.
- [8] Strużińska-Walczak A., Walczak K., *dBASE IV. Programowanie. SQL. Help*, Warszawa, 1990.

BORLAND VISUAL DBASE 7.0

The paper presents Borland Visual dBASE 7.0. Visual dBASE offers dozens of new features and language elements to provide you with a more productive, efficient work environment. This article describes the major enhancements.

**Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
„Politechnika”**

**Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
budynek A-1 PWr., tel. (0-71) 320-25-34,
„Tech”**

**plac Grunwaldzki 13, 50-377 Wrocław
budynek D-1 PWr., tel. (0-71) 320-32-52
Prowadzimy sprzedaż wysyłkową**

ISBN 83-7085-474-5