European Journal of Interdisciplinary Research and Development
Volume-07 Sep. - 2022
Website: www.ejird.journalspark.org ISSN (E): 2720-5746

DATA MANAGEMENT IN THE INSTRUMENTAL ENVIRONMENT FOR THE ACCUMULATION AND USE OF PACKAGED KNOWLEDGE

Zainab Kahtan Mohammed Abbas
Department of Computer Science, University of Diyala,
Basic Education College, Republic of Iraq
zainabkahtan@uodiyala.edu.iq

Enas Mahmood Jassim
Department of Computer Science, University of Diyala,
Basic Education College, Republic of Iraq
enasmahmoud@uodiyala.edu.iq

Abstract
The information contained in the definitions of software objects (parameters, operations, products, processors, tasks, and modules) may be of particular interest for manipulation and use. For this purpose, the SATL language suggests using special object-specifiers, or simply specifiers, in programs. The specifier may contain the same information as the software object definition, however, unlike the software object, this information is always available at the program execution stage (during the existence of the specifier as a program object), that is, it can be read, modified or used by another way.

Keyword: SATL language, SPECIFICATOR, objects.

Introduction

To implement specifiers, the following built-in data types have been introduced into the SATL language: PARAMETER_SPEC, OPERATION_SPEC, PRODUCTION_SPEC, PROCESSOR_SPEC, TASK_SPEC, MODULE_SPEC, which are the types of specifiers for parameters, operations, productions, processors, tasks, and modules, respectively. These data types can be used like C++ classes to create specific instances of specifiers and further work with them. Descriptions of built-in types in the form of C++ class definitions are not provided here and below, as they are highly implementation dependent, but the information below may help in understanding the nature of these data types as classes and in the correct use of data of these types as instances of objects of the corresponding classes. .

· The common ancestor of specifier types is the built-in SPECIFICATOR type with no-argument constructor, same-type copy constructor, virtual null single-type assignment ("="), comparison ("=="), and I/O operators, and a virtual destructor. In addition, SPECIFICATOR has function members for accessing (by reading and writing) standard attributes with appropriate types and names: char*& . . . KEY( ), char*& . . . name( ), char*& . . . COMMENT( ), TIMESTAMP& . . . LASTMOD ( ), TICON& . . . ICON ( ) [1].

Each type of a specifier of a particular type of software objects has a constructor with an argument in the form of a key of a software object of the corresponding type, with a copy constructor of the same type, operators of the same type of copying, assignment (“=”) and

comparison (“==”), input / output operators and a destructor . In addition, each specifier type has function members for read and write access to its attributes with the appropriate types and names. So for a software parameter specifier it would be bool& . . . CONST ( ), <TYPE>& . . . TYPE( ), bool& . . . DISINTEGRATE( ), where <TYPE> is implementation dependent (the definition of <TYPE> is closely related to the RTTI facility and may be expanded upon adoption of the final edition of the ANSI/ISO C++ language standard). For a software operation specifier, this would be PARAMETER_REF& . . . RETURN( ), where PARAMETER_REF is a special data type for a software parameter reference (details below), int& . . . LOGICAL( ), OPERATOR_PARMLIST& . . . PARMLIST( ), where OPERATOR_PARMLIST is a built-in class that implements a list with an ancestor LIST (details below) and member functions int SIZE( ), OPERATOR_PARMLIST_ITEM& . . . [ ] (int), where the OPERATOR_PARMLIST_ITEM class in turn has bool& function members. . . IN( ), PARAMETER_REF& . . . IN( ), bool& . . . OUT( ), PARAMETER_REF& . . . OUT( ), bool& . . . ASTERISK( ), PARAMETER_REF& . . . PARAMETER( ), bool& . . . DISINTEGRATION( ), bool& . . . EXTRAORDINARY( ), bool& . . . INTEGRATE( ). For other types of specifiers, there is a similarly constructed set of member functions for accessing the corresponding attributes.

· Built-in types of references to instances of software objects of a certain type (parameters, operations, products, processors, tasks, and modules) are named PARAMETER_REF, OPERATION_REF, PRODUCTION_REF, PROCESSOR_REF, TASK_REF, and MODULE_REF, respectively. Their common ancestor OBJECT_REF is a reference to an object of the built-in type OBJECT (depending on the implementation). OBJECT_REF defines a virtual dereference operator (“*”) to use access to the referenced object of type OBJECT. For objects of type OBJECT, an inverse virtual operator for obtaining a reference (“*”) is defined with a result of type OBJECT_REF. Operators of the same name for pairs are defined in a similar way:

PARAMETER_REF and parameter key,
OPERATION_REF and operation key,
PRODUCTION_REF and product key,
PROCESSOR_REF and processor key,
TASK_REF and task key,
MODULE_REF and module key.

· You can create a specific instance of an arbitrary software object using the special construction CREATE(<SPECIFICATOR>[<O>] [=<V>], where <SPECIFICATOR> is the software object specifier of the required type, containing the values of the attributes of the object being created, <O> – a variable of the type of a reference to an instance of the software object of the required type, <V> is the initial value, and to delete it using the special construction DESTROY (<SPECIFICATOR>) or DESTROY (<O>), which are executable statements of the SATL program.

Thus, during program execution, using specifiers and references, you can get any information about the specification of the problem area, input / output and otherwise dynamically modify it and use it to create objects of the problem area in accordance with the current state of its specification.

The ubiquitous use of such powerful tools, along with the flexibility and wide possibilities that open up, can lead to wasted resources (time, memory) at the program execution stage. To control this process, the SATL language provides special features that help the programmer, without sacrificing flexibility, to create code that is as efficient as possible. We are talking about the division of software objects into dependent on the dynamic change of other objects and independent. The corresponding optional mutually exclusive DEPENDED and INDEPENDED attributes complement the set of standard software object attributes [1].

A software object is dependent if at least one of the following conditions is true:

- the DEPENDED keyword is present in the description of this object;
- this object is created/destroyed dynamically using CREATE/DESTROY constructs;
- this object explicitly contains in its attributes a reference to another dependent object;
- at the stage of compilation of the statistical analysis of the program text, the possibility of dependence of this object on other dependent objects is not excluded, the keyword INDEPENDED is absent in the description of this object, and, in addition, compilation options do not prescribe to consider all software objects independent by default;
- compilation options prescribe to consider by default all objects of the software as dependent, and in the description of this object there is no INDEPENDED keyword.

If none of the above conditions is met, the software object is considered independent.

To increase the efficiency of your code, you should minimize the number of dependent objects in your program.

Examples of working with specifiers:

PARAMETER_SPEC ps1(p1), ps2; /* create ps1 parameter specifiers based on p1 and ps2 parameter without initial value */

ps2=ps1; /* copy the value of the specifier from ps1 to ps2 */

cout << ps2.KEY( ) << ps2.CONST( ) << OPERATOR_SPEC(01).PARMLIST.SIZE( ); /* print the values of the KEY and CONST attributes from the ps2 parameter specifier and the number of actual operation parameters 01 */

ps2.KEY="p2"; // change parameter key to p2

CREATE ps2; // and create a parameter with this key

DESTROY ps2; // destroy the parameter with the key p2

Stream I/O

For the values of software parameters and specifiers, stream input/output operators are defined, which have the notation accepted in C++ (“>>” and “<<”).

Thus, to enter from the <in> stream or output the value of the <p> parameter to the <out> stream, you must write the following: <in> >> <p> or <out> << <p>, respectively. In this case, the input stream file must contain the input value of the <p> parameter in a separate line, formatted as follows:

<p>=<v>,

where <v> is the input value formatted according to C++ rules according to the <p> value type. For arrays, intersections, structures, and classes, each element is allowed on a separate line, but the index(es) in square brackets or the qualifier(s) in the form of an element name separated by a dot must be added to the <p>.

When a parameter value is output, the output file is placed in the output file along with the parameter name in a formatted visual form suitable for putting in a report or documentation.
Specifier stream I/O operators are written similarly, but work in a slightly different way. When input, the source file must contain the specification of the software object, formatted according to SATL rules. During the output process, the specification of the software object in SATL is also placed in the output file, to which comments are added for clarity and the profiling and alignment procedure on the printed page is applied.
Stream I/O examples:
cin >> p1 >> VAR(p2) > p3; /* input values of parameters p1, p2, p3, and p2 is entered according to C++ rules */
cout << s1 << p1; /* print the value of the specifier s1 and the parameter p1 */

Knowledge bases and data

For long-term storage, fast search and updating of information in the instrumental-technological complex (ITC) SATURN [2], knowledge and data bases are used. Based on the fact that the division into knowledge and data bases is very conditional (it is believed that knowledge bases in their pure form should be used only for storing software specifications, databases in their pure form - only for storing calculated data), and also for practical reasons in ITC SATURN, any database is capable of storing both, in the future we will simply call them databases (DB).
The SATL language uses a built-in data type called BASE to work with the database. The following are defined as function members for the BASE class:
- a constructor for opening a database with an argument in the form of a string containing the path to the corresponding file;
- copy constructor of the same type;
- same-type assignment operator (“=”);
- comparison operator (“==”);
- PATH function to get a string containing the path to the database file;
- destructor for closing the database.

As a result of compiling and assembling a program in the SATL language, a target database of the package (knowledge base) is created that contains all the specifications of the software objects defined in the program. The built-in value MAIN of an object of type BASE corresponds to this database. The built-in DATA value of the BASE type object corresponds to the database for automatically saving parameter values. A built-in value named NONE of objects of type BASE means no database.

Database operation examples:

BASE b1("c:\\p\\b1"); /* open the database from c:\p\b1 directory */
BASE b2=b1; /* duplicate b1 into b2 */
if (b1==b2) . . . /* compare b1 and b2 */
count << b2.PATH; /*output the path for b2*/
~b1; /* close b1 */

Closely related to objects of type BASE are objects of the built-in type SET - sets or sets. The following are defined as function members for the SET class:

constructor without arguments;
constructor with arguments OBJECT& = ALL_PARAMETERS, BASE& = NONE, char* = "1" (value option);
constructor with arguments BASE& = NONE, char* = "1" (value option);
constructor with arguments SET&, BASE& = NONE, char* = "1" (value option);
constructor with arguments LIST&, BASE& = NONE, char* = "1" (value option);

Constructor of the same type of copying;
operators of the same type assignment ("=") and comparison ("==");
Operators of the same type multiple addition ("+") and subtraction ("-"), which are actually performed on sets of software objects, on lists and variants of values specified as arguments in the constructors of the corresponding operands (databases are only summarized);
· single-type input (output) operator: ">>" ("<<") with a result of SET type, copying the intersection of two sets from the objects specified by the left operand to the objects specified by the right operand (and vice versa). In this case, objects are taken from the database if the operand contains a database, or from RAM if the operand does not contain a database;
input (output) operator: ">>" ("<<") with the second operand of the SPECIFICATOR& type and the result of the SET type, copying the specification of the object specified by the left operand to the specifier specified by the right operand (and vice versa);
· Functions for accessing set elements: SET_PARMLIST& . . . PARMLIST( ), where SET_PARMLIST is a built-in class with an ancestor of type LIST and function members: int . . . SIZE( ) and PARAMETER_REF& . . . [ ] (int) for parameters, similarly constructed functions for accessing software objects of other types SET_VARILIST . . . VARILIST( ), where SET_VARILIST is a built-in class with LIST ancestor and function members: int . . . SIZE( ) and char*&. . . [ ] (int) for value options, SET_BASELIST . . . BASELIST( ), where SET_BASELIST is a built-in class with an ancestor of type LIST and function members: int . . . SIZE( ) and BASE& . . . [ ] (int) for the database; destructor.
The SET type has a set of built-in values: ALL_PARAMETERS, ALL_OPERATIONS, ALL_PRODUCTIONS, ALL_PROCESSORS, ALL_TASK, ALL_MODULES, ALL_VARIANTS, ALL_BASES are the sets of all software parameters, software operations, products, processors, tasks, modules, options and databases, respectively, ALL is the set of all listed sets, except for the database. The special built-in value DELETE is used in the "<<" and ">>" operations to delete.

Examples of working with sets:
SET(b1, v1) >> ALL_PARAMETERS; /* copy the values of option v1 of all parameters from database b1 to parameters in RAM */
SET(p1, b1, v1) << DELETE; /* delete option v1 of parameter p1 from database b1 */
SET(o1, b1) >> o_spec1; /* copy software object specifier o1 to specifier o_spec1 */

SET(ALL, b1) >> SET(ALL, b2) >> SET(ALL_TASK, b3); /* copy content from database b1 to database b2 and all tasks from database b1 to database b3 */

SET(PROCESSOR_SPEC(proc1).PARMLIST-p1,b1,v1) >> ALL_PARAMETERS; /* copy the value of the v1 variant of all parameters of the proc1 processor, except for p1, from the database b1 to the parameters in RAM */

SET(p1+p2, b1) << DELETE; /* remove specification of parameters p1 and p2 from database b1 */

The SATL language contains special tools for automating the process of writing parameter values to the database. Using the optional attribute SAVE [(<SAVE>)], added to the attributes of a software parameter, you can specify as <SAVE> an object of type SET with a specific database and the name of a variant to automatically save the value of this parameter. By default, <SAVE> is taken as SET(DATA, "1"). Software options that do not have the SAVE attribute can be set to be automatically saved by default using compiler options.

With the help of a special construction (executable statement) SAVE one or several places in the program are specified where the parameter values will be automatically saved according to their attributes and compilation options.

Example:

PARAMETER KEY(p1) TYPE(. . .) SAVE;
PARAMETER KEY(p2) TYPE(. . .) SAVE (SET(b1,"2");
PARAMETER KEY(p3) TYPE(. . .);
. . .
SAVE;
. . .

Here, when executing the SAVE statement, the value of the p1 parameter will be automatically saved in the DATA database with the option name "1", the p2 parameter in the b2 database with the option name "2". Automatic saving of the p3 parameter value will be performed depending on the compiler options.

Specialized object-oriented data model (SOOMD)

The choice of data model is a key issue in an efficient SATL implementation. First of all, we are talking about the conceptual scheme of software specification in ITC SATURN [2], a general idea of which is given in [1], and a more detailed presentation is available in [3]. The adequacy of the representation of the software conceptual scheme is the main criterion for choosing the required data model.

As a result of the detailed analysis carried out in [3,4], it was revealed that the required data model must belong to the class of object-oriented models and satisfy a number of requirements that characterize its specialized nature.

Requirements for SOOMD

1) The model must be strongly typed: each object must belong to a specific data type.

2) Each object must have a "foreign key" - uniquely identifying its value, by which the system provides the fastest access to the object. It is necessary that this value does not change during the entire lifetime of the object. It is desirable that it is not part of the object.

3) The means used to describe the type should be at the level of OOP languages such as C++ (templates, multiple inheritance, virtual methods/ancestors, event management, exceptions) in their expressive power. For variant records, automatic selection of the actual variant can be used. Variable length fields must be supported.

4) Objects can be combined into lists, which in turn are objects of the corresponding types. Lists can be iterated in any direction. Quick search in lists should be provided by predefined key expressions. The search can be carried out by several key values at once. Some key expressions may be declared unique, and the system must support this as an implicit integrity constraint.

5) Values can have multiple representations. The system should contain built-in facilities for input/output and adjustment of values in various representations, depending on which of them is most relevant to the needs of the user. When entering and correcting values, an intensional prompt can be used, a choice from an extensional list of valid values based on existing analogues or according to history.

6) Objects can enter into n-ary relations (connections). Each specific relationship must belong to a specific type. The type of connection determines the types of objects that act in this connection in the i-th role, the minimum and maximum cardinal number, integrity constraints imposed on the objects participating in the connection. Link types can be defined by inheriting the properties of previously defined link types.

7) The link support mechanism (presentation, navigation, correction, automatic tracking of integrity constraints) should be built into the system.

8) Types are objects of type “type”. They can be created, modified, and deleted during the session (the corresponding transition to the slow interpretation mode should only be done when necessary).

9) Computational relations can be defined over the types of objects, indicating the role of objects (input/output), implemented by the corresponding functions. With the use of computational relations, automatic synthesis of programs should be carried out.

10) The lifetime of an object should not be limited to one session. Physically, an object can be placed in a special long-term "storage" - space. Multiple spaces can be accessed at the same time. To preserve the integrity of links between spaces, an automatically performed procedure for updating data based on version-time dependencies should be provided.

11) The database schema should be stored along with other data in a database with open access to it for the purpose of reading or modifying.

The implementation of the conceptual scheme of software specification within the SOOMD, which meets the above requirements, is, in comparison with implementations by means of traditional models, completely transparent, requires minimal programming, and is most effective when solving practical problems with large volumes of software specifications.

SOOMD specifications

When describing the SOOMD specifications, we will focus, as far as possible, on the terminology used in [5], and in terms of language constructs, on the C++ programming language [6].

A schema within the framework of a COOMD is a set of class definitions over the alphabet B, where B consists of a set of class symbols C, a set of attribute symbols A (used in defining the class structure), a set of symbols of n-ary relations over classes U, and a set of method symbols M. Next we will use the symbols c, a, u, m as elements of the sets C, A, U, M, respectively.

Let's assume that some set of base classes (integer, real, strings, etc.) is initially built into SOOMD.

Each class is characterized by the value type of the objects of that class. The type declaration t is constructed as follows:

t ::= c | u | <t> : t1, ... tn | <etc_in_C++_style> |
select by(i) <t> {t1 a1; ...: tn an} |
enum <t> {<x1> a1; ...: <xn>an} |
t1 <t> [n1, ..., nn] |
blob<t> ,

where <t> is the type of the type being defined; t1, ..., tn are previously defined types; <etc_in_C++_style> – C++ style type definition; i and n1, ..., nn are previously defined attributes of an integer type or constant; <x1>, ..., <xn> are optional representations of enumerated values. Complex structures are interpreted as follows: select – variant notation with automatic selection of the actual variant by value i; enum - an enumerated type with the ability to define representations of enumerated values; t1 <t> [n1, ..., nn] – array with boundaries n1, ..., nn and elements of type t1; blob is an arbitrary binary object with no predetermined size.

The definition of method m has the following form:

method m (c1, ..., cn) returns (c1¢, ..., cn¢) ... ,

where c1, ..., cn is the input of the method; c1¢, ..., cn¢ is the output of the method. Additionally, we will assume that SOOMD has tools for defining and implementing methods that are comparable in capabilities to a language such as C ++ (operator overloads, virtual methods, etc.).

The class definition c has the following form:

class c type (t) methods (m1, ..., mn) ,

where t is the value type of objects of class c; m1, ..., mn are class methods.

The definition of the relation u has the following form:

relation u (c1(n1, n2), ..., cn (n3, n4))
[keys (k1, ..., kn)]
[constains (constr1, ..., constrn)] ,

which is interpreted as the definition of an n-ary relation over the classes c1, ..., cn, where n1, n2 are cardinal numbers, which mean, respectively, the minimum and maximum number of objects of class c1 associated with objects of classes c2, ..., cn, n3 , n4 are cardinal numbers, which mean, respectively, the minimum and maximum number of objects of class cn associated with objects of class c1,..., cn-1; k1, ..., kn are expressions from the attributes of the classes c1,

..., cn, which form the keys of the relation u (unique keys are marked with the unical prefix); constr1, ..., constrn is an integrity constraint imposed on instances of the classes c1, ..., cn participating in relation u.

Objects and foreign keys. Instances of classes and relationships are objects. There are external and internal objects.

External objects include instances of classes and relationships that are not part of larger (enclosing) objects. Internal objects include instances of classes and relations that are part of larger (encompassing) objects and are created together with them as their constituent parts.

Each instance of a class/relationship is automatically assigned a unique foreign key upon creation (not part of the structure of this class/relationship). The foreign key allows each external instance to be uniquely identified and accessed regardless of its location, as well as the internal instance to be accessed within the enclosing object. A foreign key is a data structure of the EXKEY type, which can be included as an attribute of the corresponding type in other objects/relations. The foreign key contains information about the location of the object and its version (for synchronization with distributed storage of objects).

The following operations are performed on foreign keys (C++ style syntax):

new - create an instance of an external object and return its foreign key;

& - get the foreign key of this object;

* – get access to an external object using this foreign key;

:> – get access to the internal object by the given foreign key;

–> – get access to an element of the external object structure by the given foreign key;

base - access the database instance using the given foreign key;

++ is the next instance of this class/relation;

-- is the previous instance of this class/relation;

– get the foreign key by the number of the object of the given class/relation;

count - get the total number of objects of the given class/relation.

Built-in objects. COOMD initially has a set of built-in classes (type class cT, class class cC, relationship class cR, attribute class cA, method class cM, object class cO) and built-in relationships between them.

The type class cT defines a set of types T. All data types (t) are objects of the class cT.

The class class cC defines a set of classes C. All classes (c) are objects of the class cC. All types, relations, attributes, methods are objects (instances) of the corresponding classes.

The CC class class has a built-in create method for generating class instances (including instances of the CC class class):

method create (cC, cT) returns (cO),

where the source class c1 must be specified as an argument of the cC class; as cT - initial values; as sO - the resulting instance o1 of class c1. To create an instance of the c1 class, the appropriate constructor method of the c1 class is used, and the create method allows you to dynamically set the class of the object being created.

Accordingly, there is a built-in destroy method:

method destroy (c0) returns ( ),

which also calls the appropriate destructor method of class c1 if an object of class c1 is passed as cO.

The built-in ca11 method allows you to call arbitrary methods of an arbitrary class (including create, destroy, call of the CC class):

method call (cM, cO, cT) returns (cT).

The built-in relation above classes hierarchy defines the class hierarchy:

relation hierarchy (cC(O, MAXPARENTS), cC (O, MAXCHILDREN))...

The class of relations cR defines a set of relations R. All relations (r) are objects of the class cR. The cR relation class has a built-in create method to build a new relation r based on the old r1:

method create (cR, cT) returns (cR),

where the source class r1 must be specified as the cR argument; as cT, the filter value; as cR, the resulting new ratio r. Thus, the fastest search with the maximum use of keys is carried out.

The built-in count method allows you to find the total number of instances of a given relationship, and operations on foreign keys make it easy to operate access to instances of relations.

In addition, a full set of operations of the relational data model is performed on relations (see, for example, [7]) using the corresponding built-in methods: boolean, selection, projection, join, division, equijoin, splitting, factor, and relational algebra operations (union, intersection, difference, active complement, choice, natural join, division, rename, psi join).

The attribute class cA defines a set of attributes A. All attributes (a) are objects of class cA.

The class of methods CM defines a set of methods M. All methods (m) are objects of the class CM.

The CO object class defines an abstract object class at the top of the property inheritance hierarchy (so that all other objects have the CO object class as an ancestor). The CO object class has built-in methods new and ~ for creating and destroying objects, and operations on foreign keys make it easy to manipulate access to objects. The CO object class has an isA attribute of type EXKEY to indicate the class to which the object belongs.

The built-in object class sV defines a set of DB objects (independent distributed information storages). Each object is stored in a specific database.

The description of class cB is as follows:

class cB (contents) methods (open, close, synchronize, ro, rw, sawe, autosave, <controls>),

where contents is a relation containing information about the contents of the database; open/close – methods for opening/closing the database; synchronize - a method that provides version-time synchronization of the database; ro/rw – methods that set database access modes (read/read and write only), save/autosave – methods that provide database saving modes; <controls> - other methods for performing service functions with the database.

The description of the contents relation is as follows:

relation contents (cC(1, MAXCLASS), co(0,MAXOBJECTS))...,

which makes it easy to access the contents of the database.

The built-in base method makes it possible to access a database instance by a given foreign key, other operations on foreign keys make it easy to operate on the contents of the database.

Planning and automatic synthesis of programs. When implemented within the framework of SOOMD ITC SATURN [2], object classes can be declared as parameters, methods - as

operations of the problem area. Thus, the possibility of planning and automatic synthesis of programs over SOOMD objects is realized.

References

1. Zhuravlev A.E., Oparin G.A., SATL - a language for the presentation and use of package knowledge // Software products and systems. - 1996. - No. 3. - S. 26–35.
2. Oparin G.A., Feoktistov D.G., Zhuravlev A.E. Tools for building and operating hybrid computing-type expert systems // Computational technologies. - Novosibirsk: ICT SO RAN, 1993. - T. 2, T. 6. - P. 13–26.
3. Zhuravlev A.E., Oparin G.A. Representation of knowledge about the problem area in the instrumental-technological complex SATURN / PZ. Conceptual scheme and choice of data model // Computer logic, algebra and intelligent control. - Irkutsk: IrVTs SO RAN, 1994. - T. 1. - P. 76–88.
4. Zhuravlev A.E., Oparin G.A. Specialized database management system in the SATURN-ES instrumental system // Intellectualization of programming tools, 1990.
5. Tsikritzis D., Lochowski F. Data Models. - M.: Finance and statistics, 1985. - 344 p.
6. Stroustrup B. The C++ Programming Language. Second Edition, Addison-Wesley Publishing Company, 1991. - 669 p.
7. Meyer D. Theory of relational databases. – M.: Mir, 1987. – 608 p.