Test data generation for covering functionality of database applications

Evgeny Kostychev, Vitaly Omelchenko, Sergey Zelenov
Institute for System Programming
at the Russian Academy of Sciences
Moscow, Russian Federation
e-mail: {kostychev, vitaly, zelenov}@ispras.com

Abstract—**Applications for processing great volumes of data is a very widely used kind of software. In enterprise integration there are tasks of data integration. When solving these tasks, special tools supporting development and execution of applications implementing extract, transformation and load pattern are often used. From the point of view of functional testing, such applications have a specific peculiarity related to a huge number of combinations of input data. Existing approaches and tools solving the problem of test data generation for database application build large arrays of input data based on database scheme or on SQL queries of application under tests. To ensure covering functionality of an application under test using these approaches and tools, a brute force of all available combinations is needed. In the paper, we prpose a method allowing less excessive data generation for covering functionality of database applications. It allows achieving functionality coverage with acceptable amount of test data close to optimal one (one test per one functionality branch) in acceptable generation time.**

Test data generation; database applications; functional testing; data integration; ETL-applications testing

I. Introduction

Nowadays, availability of more and more increasing volumes of information storages and computing resources for their processing is constantly growing. This yields prevalence of applications working with huge amount of data in various areas. For instance, in the early nineties of the last century, the British national corpus has been created. This is a specially marked and processed big set of printable and audio texts containing 100 million words for more than twenty years' interval [1]. Since then, such national corpora of many languages have appeared and continue to appear and develop. The corpora not only make many traditional problems of linguistics more trivial, but also allow to state and to solve previously impracticable problems primarily related to processing of great volumes of data [2]. Also problems concerned with processing of great volumes of data rise in such areas as statistics, sociology, geophysics, some sections of physics, molecular genetics, problems with climate, meteorology.

Storage and processing great volumes of data is crucial for modern business too. Enterprise systems contain huge volumes of interconnected data concerning consumers and customers, suppliers, partners, equipment and personnel, financial streams, various business transactions. Huge volumes of data are usually stored in structured form in one or several databases under control of one or several DBMS. Often, supporting various business needs require an integration of several subsystems of an enterprise system or even connecting to some parts of external systems. In many such cases, a problem of access to and operating with data stored in different databases in various formats rises. Therefore, data integration applications that perform replication, updating and synchronization of great volumes of data form a widespread kind of integration applications.

Tasks of such applications can be simple themselves but not always easy for implementing (for example, selection of data concerning some person or legal entity). The tasks can also be enough difficult both in conditions of selection/aggregation of input data and in calculation of final result (for example, invoicing clients according to volume of actual service consumption, tariff plans, and provided discounts). Since this kind of tasks frequently occurs when solving the integration problems, there are specialized platforms providing universal (with respect to platforms for data storage) environment for development and running integration applications that effectively solve problems of extraction, transformation and loading of big data [3]. Further we refer to such applications as ETL-applications (Extract, Transform and Loading).

After development or updating of any application, one should check correctness of its behavior with respect to the functional requirements implementing the business needs. The business logic of ETL-applications can be detailed as follows:

- extraction source data that meet corresponding conditions defined in requirements;
- changing values of source data with respect to conditions defined in requirements;
- transformation of input data (changing values and/or representation format) according to requirements;
- loading into target some data that should be loaded according to requirements;
- changing in the target some data that meet corresponding conditions defined in requirements.

Mainly, one checks correctness of implemented behavior by means of functional testing. That is running the system under test (SUT) on specially prepared input data and comparing output data with expected ones. In the paper, we consider the problem of generating input data for functional testing of ETL-applications. Note that ETL-applications contain all the same steps as any database applications (data extraction, transformation and loading). Therefore, proposed method is applicable not only to ETL-applications but to many other database applications too.

One can estimate quality of functional testing by achieved coverage of functional requirements. Ideally, all functional branches of the algorithm implementing required functionality should be covered. To perform that, one should provide such test data that force SUT to do the following:

- retrieving data from the source and filtering them on all possible combinations of conditions specified in the requirements;
- covering during data transformation all possible functional branches of the algorithm implementing the required transformation;
- possibility to check the absence of unnecessary changes of source and target data;
- processing special data (empty columns, lines of limited length, etc.).

As a rule, a test is defined by some parameters. Obviously, all possible combinations of parameters contain combinations needed for covering the functionality of the application under test. But in real situations, when generating all possible combinations, the number of parameters to be combined causes combinatorial explosion. In this case, besides of unacceptable generation time and total amount of test, resulting test set contains many different test data values not distinguishable for SUT functionality. A problem of analyzing test results rises: the same errors may be revealed on thousands of different values of test data, and thus, time-consuming analysis is required.

Ideally, for covering functional requirements, test data should provide exactly one possible combination of Boolean expression for each functionality branch. The paper describes a method of generating test data with volume close to the optimal set.

The rest of this paper is organized as follows. Section II contains some preliminaries. Section III reviews related work. Section IV describes proposed method. Section V illustrates the proposed method by example. Section VI discusses benefits of the proposed methods. Section VII concludes the paper.

II. Preliminaries

In general, to ensure full coverage of all functional branches, one should use brute force combination of field values depending on which branching should occur when retrieving, when running an application under test. But it leads to the combinatorial explosion causing unacceptable cost on generation time and total volume of test data.

Often, in terms of functionality coverage, values of several fields are related by some semantics, which means that value of one field depends on values of some other fields. As a rule, iterating only the combinations of values meeting this semantics greatly reduces the total number of variants. Let us consider the following example. Let fields `A` and `B` take integer values in the range `[1 .. 30]` and

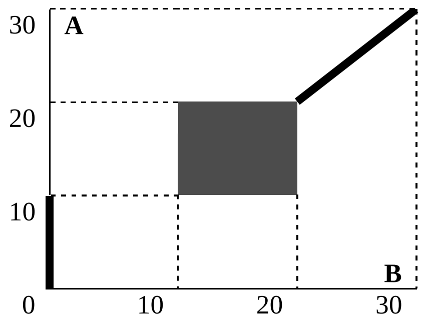

Figure 1. Optimazation of iterating by semantics.

value of the field `B` depends on value of the field `A` as follows (see Figure 1):

- if $1 \leq A \leq 10$, then $B = 1$;
- if $11 \leq A \leq 20$, then $11 \leq B \leq 20$;
- if $21 \leq A \leq 30$, then $B = A$.

When producing not all the possible combinations of values of the fields, but only ones meeting the above conditions, the amount of derived combinations is about 8 times reduced (from 900 to 120).

Besides, the functionality of an application under test can have several independent aspects. In this case, it is possible to reduce the number of combinations using so-called diagonal combinator iterating a set of tuples S such that for any i ($1 \leq i \leq n$) and for any s from S_i there exists a tuple ($s_1,\ldots,s_n$) from S with $s = s_i$, where S_i is a set of values iterated by i-th subiterator. This method ensures producing a set of test data containing every value of each field. The main advantage of this combination method is that the volume of result test data is significantly less than in the case of using Cartesian product, since the capacity of produced set equals the maximum of capacities of value sets of combining fields instead of product of them.

III. Tools Review

Most data generation tools for DB (DTM Data Generator [4], Turbo Data [5], DBMonster [6]) support filling the database tables by a large number of syntactically correct data and provide the following set of features:

- random data generation with ability to specify intervals of numeric types, length of string type, and data format;
- data generation from the list of values with ability to specify the percentage of each list row in generated data set;
- generation by selecting data from specified tables;
- generation by selecting data from files;
- generation of auto incremental data with specified initial value and step;
- data generation from existing libraries;
- random data generation by a mask;
- data generation for dependent tables;
- data generation based on external processes of generation specified by user procedures.

Some tools (AGENDA [7], HTDGen [8]) support the generation of data not only on the basis of constraints defined by schema or user, but also based on SQL queries of database application under test. It allows generating such data that SQL queries of an implementation under test return some meaningful results. However, when generating data is based on an implementation, there is no guarantee of covering all functional branches because the implementation may contain wrong branches or does not implement the required ones. Also this approach does not allow covering all required functional branches of transformation and filtering.

In order to achieve full coverage of all functional branches using any of mentioned above tools a tester has to use brute force combination of field values, depending on which an application under test extracts, transforms and filters data.

The Pinery test generating tool developed at ISP RAS [9, 10] is intended to generate structurally complex test data. Generating data having some syntactic structure is managed by specifying constraints on desired data fragments in the terms of the structure. In particular, Pinery supports so-called conditional constraints that should be used if generation of some part of data depends on values of some other parts of the generated data.

In Pinery, there are many various ways to specify constraints on values of fields. Some of them are:

- enumerating a list of desired values;
- defining a function depending on values of other fields;
- specifying a set data to be belong, for example:
  - a segment of integers;
  - a set of values of another field (e.g., in the case of assignment of values for a foreign key).

Another important kind of constraints is specification of way to combine values of fields. In particular, it is possible to build a hierarchy of different combinators containing Cartesian products, diagonal combinators and custom combinators based on dependences between values of different fields.

These features allow a tester to customize generation more exactly and, as a result, to receive test data with volume close to optimal one.

IV. Method Description

The method of directed generation of test data is based on UniTESK [11, 12] approach to model-based testing and consists of the following phases.

The first phase is requirements elicitation: analytics study normative documents for the system under test, identify input data requirements and categorize them. The result of the phase is a requirements catalogue that contains precisely formulated input data requirements, classified into several groups with established links between them. The catalogue is used on the following phases.

The second phase is formalization of requirements. Elicited input data requirements get specified using appropriate formal notation. Such specification is called formal model.

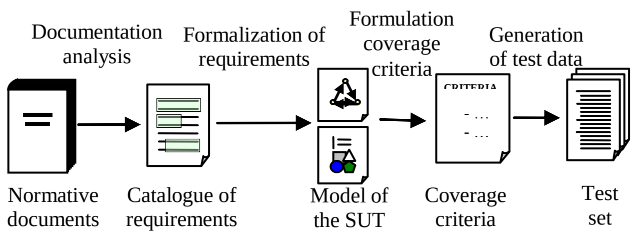

Figure 1. Phases of model-based testing.

The third phase is formulation coverage criteria: the input data domain divides into finite number of subsets (possibly intersecting each other), such that for each subset S, behavior of the SUT on any data from S is uniform. The starting point for such a division is a set of conditions from the requirements catalogue. Besides, the specific semantics of these conditions may force a tester to formulate some additional hypotheses on behavior of the SUT that induce a subdivision of the input data domain into smaller subsets.

The fourth phase is automated tests generation from the formal data model with respect to the coverage criteria. In our approach, automated generation is carried out using the Pinery generator tool. Input data for Pinery are:

- formal description of the test data model, and
- generator configuration aimed at achieving the coverage criteria.

Tests running, reports analysis, defects identification and corrections is beyond the scope of this article. These issues are not discussed here.

V. Example

Let us illustrate the proposed method by the following example of test data generation for testing simplified bank credit system.

- The normative document is the following informal description of the bank credit system.
- The client of certain type registers in the system and receives an identifier.
- The client gets a sum of money as a loan that should be repaid in few months.
- The client should monthly repay the sum calculated as current debt divided by quantity of months before the loan termination.
- If actually repaid sum is less than the calculated sum of monthly payment, then the client can be penalized. If actually repaid sum is greater than the calculated sum of monthly payment, then the client can get a bonus. Sizes of penalties and bonuses depend on both client type and credit type.
- The client can declare to get monthly e-mail notifications about the state of the loan, repayments and penalties or bonuses.
- At the end of each month, the application does the following:

A. it reduces the debt by the sum of the client repayment and calculate the penalty or the bonus with respect to the following rules.

I. If the sum of client repayment is less than monthly payment, then:
   1. clients of the type "VIP" should not be penalized;
   2. clients of the type "USUAL" should be penalized with the sum equal to 3% of the overdue payment.

II. If the sum of client repayment is not less than the monthly payment and not more then the current debt, then:
   1. clients of the type "VIP" with the loan of the type "B" and clients of the type "USUAL" with the loan of the type "A" should get bonuses equal to 2% of the repaid sum;
   2. clients of the type "VIP" with the loan of the type "A" should get bonuses equal to 5% of the repaid sum;
   3. clients of the type "USUAL" with the loan of the type "B" should get bonuses equal to 1% of the repaid sum.

III. If the sum of client repayment is more than the current debt, then the bank personnel should be notified about refund.

B. If the client has declared monthly e-mail notification, then the e-mail should be sent.

At the first phase, we analyze the normative documents and build a requirements catalogue. In this example, the requirements catalogue consists of the items of the bank credit system behavior rules (both A with sub-items and B). They can be represented in the form of the diagram (see Figure 3).

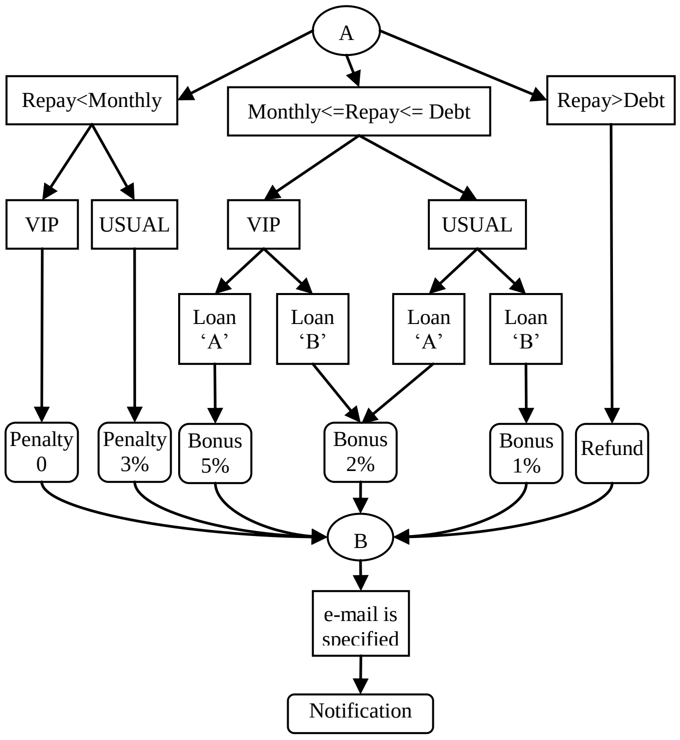

Figure 2. Diagram of bank credit system requirements.

TABLE I. BEHAVIORIAL PARAMERTERS OF BANK CREDIT SYSTEM

Short Name	Full Column Name	Type	Description
MP	MONTH_PAYMENT	NUMBER	Sum to be pay monthly
D	DEBT	NUMBER	Total debt sum
P	REPAYMENT	NUMBER	Actually repaid sum
CL	CLIENT_TYPE	CHAR	Client type: "V" – VIP client "U" – USUAL client
CR	CREDIT_TYPE	CHAR	Credit type: "A" – type "A" "B" – type "B"
E	EMAIL	STRING	E-mail address

At the second phase, we build the formal model of the requirements.

The behavior of the application under test depends on the parameters presented in TABLE I.

The constraint of data consistency is `MP <= D`.

The requirements can be formalized as follows:

```
A.I     P < MP && CL = "V" => penalty 0;
        P < MP && CL = "U" => penalty 3%;
A.II.1     MP <= P <= D
        && (   CL = "V" && CR = "B"
            || CL = "U" && CR = "A"
           )
        => bonus 2%;
A.II.2     MP <= P <= D
        && CL = "V"
        && CR = "A"
        => bonus 5%;
A.II.3     MP <= P <= D
        && CL = "U"
        && CR = "B"
        => bonus 1%;
A.III   P > D => refund.
B.      E ≠ "" => notification.
```

At the third phase, we formulate coverage criteria.

Besides the requirement conditions, we introduce the following additional hypotheses about behavior of the application under test:

1. If `MP <= P <= D`, then behavior of the application can differ in the following cases:
   - `MP = P = D`;
   - `MP = P < D`;
   - `MP < P < D`;
   - `MP < P = D`.
2. If conditions of the requirement A.I hold, then behavior of the application can differ in the following cases:
   - `CL = "V" && CR = "B"`;
   - `CL = "U" && CR = "A"`.

Here we proceed with establishing rules for division the input data domain into subsets with uniform behavior of the application. First, we formulate such rules separately for each parameter, and then we state how to combine these rules for the whole input data domain.

Let us establish division rules for parameters `MP`, `D`, `P`, `CL`, and `CR`, that relate to requirements from the A group.

The hypothesis 1 yields the following division of input data for parameters `MP` and `D` into two subsets:

- `MP = D`;
- `MP < D`.

Conditions from the requirements of the A group yield the following division rules for parameter `P`.

- If condition `MP = D` holds, then domain for parameter `P` divides into three subsets:
  - `P < MP`;
  - `MP = P = D`;
  - `P > D`.
- If condition `MP < D` holds, then domain for parameter `P` divides into five subsets (taking into account the hypothesis 1):
  - `P < MP`;
  - `MP = P < D`;
  - `MP < P < D`;
  - `MP < P = D`;
  - `P > D`.

Conditions from the requirements of the A group yield the following division rules of input data for parameters `CL` and `CR`.

- If condition `P < MP` holds, then conditions from the requirements of the A.I group yield division into two subsets corresponding to all possible values of the parameter `CL`.
- If condition `MP <= P <= D` holds, then conditions from the requirements of the A.II group and the hypothesis 2 yield division into four subsets corresponding to all possible combinations of values of parameters `CL` and `CR`.
- If condition `P > D` holds, then the condition from the requirement A.III yields no division (or, formally speaking, division into one set).

Next, let us establish division rules for the parameter E that relate to the requirement A.II. The condition from the requirement A.II yields division into two subsets: with empty and non-empty value of `E`.

Next, let us state how to combine these rules for the whole input data domain.

Division of input data domain related to parameters of A group (`MP`, `D`, `P`, `CL`, and `CR`) is induced by Cartesian product (taking into account all dependencies) of divisions for each of the parameters.

Since behavior of the application under test has two independent aspects A and B, then division of the whole input data domain is induced by the diagonal combination of the divisions corresponding to A and B.

At the fourth phase, we generate tests automatically using the Pinery generator tool.

First, we should provide Pinery with formal description of DB scheme (for example, using the DDL-subset of SQL). Next, we should configure Pinery by constraints on data to be generated. These constraints are described in terms of the DB scheme elements.

In our example, there is one table CREDITS with the following fields: MONTH_PAYMENT, DEBT, PAYMENT, CLIENT_TYPE, CREDIT_TYPE, EMAIL. Further we refer these fields by their short names.

In this example, there are two kinds of constraints:

- Constraint on values of one field;
- Constraint on combination method for several fields.

First, we describe constraints of the first kind.

In order to cover subsets with `MP = D` and `MP < D`, we may put, for example, `MP = 6`, `D = 6` and `D = 30`. In order to cover subsets with empty and non-empty E, we may put, for example, `E = ""` and `E = "…@…"` (some address).

In order to configure Pinery with these values, we should describe the following constraints that enumerate lists of values for each field[1]

```
MP = { 6 };
D = { 6, 30 };
E = { "", "…@…" };
```

These are examples of so-called unconditional constraints that specify values valid in all cases.

However, sometimes we must not use unconditional constraints. For example, values of the field P depends on values of fields `MP` and `D`. Thus, we should use two conditional constraints: for the cases `MP = D` and `MP < D`.

In order to make condition `P < MP` hold, we may put `P = MP − 1`. In order to make condition `P > D` hold, we may put `P = D + 1`. If condition MP <D holds, then in order to make condition `MP < P` <D hold, we may put `P = (MP + D)/2`. As a result, we have the following constraints for the field `P`:

```
P[MP<D] =
    {MP-1, MP, (MP+D)/2, D, D+1};
P[MP=D] = {MP - 1, D, D + 1};
```

Values of fields `CL` and `CR` depends on values of fields `P`, `MP` and `D`:

```
CL[ P<MP ] = { "V", "U" };
CR[ P<MP ] = { "A" };
CL[ MP<=P && P<=D ] = { "V", "U" };
CR[ MP<=P && P<=D ] = { "A", "B" };
CL[ P>D ] = { "V" };
CR[ P>D ] = { "A" };
```

Here we proceed with description of a combinator of fields values for generation of CREDITS table tuples.

In all cases, we may combine values of fields `CL` and `CR` using Cartesian product:

```
Product( CL, CR )
```

Similarly, we may combine values of fields `MP` and `D`:

```
Product( MP, D )
```

[1] In Pinery, constraints are described in XML form. In order to increase readability, here we describe constraints in a semi-formal pseudo-code.

Since values of fields `CL` and `CR` depend on values of fields `P`, `MP`, and `D`, and values of field `P` depend on values of fields `MP` and `D`, then we should use special "dependent" combinator that describes combinations of the fields that relate to requirements from the A, B and C groups:

```
Depend(  Product( MP, D )
      => P
      => Product( CL, CR )
      )
```

As we mention above, we should use diagonal combinator to combine fields that relate to requirements from groups A and B. So, we have the following combinator for generation of CREDITS table tuples:

```
combinator( CREDITS ) =
   Diagonal( Depend(  Product(MP, D)
                   => P
                   => Product(CL, CR)
                   )
           , E
           );
```

Resulting test data is presented in TABLE II.

TABLE II. GENERATED TEST DATA

D	MP	P	CL	CR	E
6	6	5	V	A	
6	6	5	U	A	@
6	6	6	V	A	
6	6	6	V	B	@
6	6	6	U	A	
6	6	6	U	B	@
6	6	7	V	A	
30	6	5	V	A	@
30	6	5	U	A	
30	6	6	V	A	@
30	6	6	V	B	
30	6	6	U	A	@
30	6	6	U	B	
30	6	18	V	A	@
30	6	18	V	B	
30	6	18	U	A	@
30	6	18	U	B	
30	6	30	V	A	@
30	6	30	V	B	
30	6	30	U	A	@
30	6	30	U	B	
30	6	31	V	A	@

VI. DISCUSSION

Using of a dependent combinator allow us to have a uniform configuration of the generator instead of two (for cases `MP = D` and `MP <D`).

Using of a dependent and diagonal combinators allow us to reduce quantity of generated test data by more than 65% in comparison with the general Cartesian product: We have 22 tuples while Cartesian product gives 64 tuples (`3*2*2*2 = 24` for case `MP = D`, plus `5*2*2*2 = 40` for case `MP < D`). Nevertheless, our test set has the same quality as the Cartesian product test set (with respect to the formulated coverage criteria).

There are two aspects that make relative reduction of tests quantity to increase.

First, the more possible values of fields are, the more economy we have. For instance, if we have in our example one addition client type and one addition credit type , then economy in our approach is 70% (`3*3*3*2 + 5*3*3*2 = 144` for Cartesian product against `(3 + 1*3*3 + 1) + (3 + 3*3*3 + 1) = 44` in our approach).

Second, the more independent aspects of behavior of the application under test, the more economy we have. Suppose in our example, that actual payment has a type with two possible values ("O" – payment under the clearing settlement and "E" – payment by cash), and the application under test uses this type in some additional aspect of behavior (for example, calculating some statistics). Then the quantity of tuples in our approach does not increase (since we use diagonal combinator), while the quantity of tuples for Cartesian product doubles.

VII. CONCLUSION

In the paper, we propose the method of automated generation of test data for functional testing of applications that process huge volumes of data. The method is aimed to cover functional branches of an application under test. The main benefit of the method is that on the one hand it allow a tester to achieve coverage of functionality of an application under test, but on the other hand generated test data are more optimal then in existing tools:

- generation process is less time-consuming, and
- test report analysis is less labor-consuming and less time-consuming.

Generation of the data by means of Cartesian combination of all fields provides full coverage, but resulting test data are practically always superfluous. Redundancy extremely grows under increasing number of combined fields and cardinality of sets of their values, on which the behavior of the application under test depends.

The proposed approach is based on requirements analysis and formalization. Usage both Cartesian product combinators, and dependent and diagonal combinators allows a tester to reduce a test set without loss of test data quality and to obtain test data with volume close to the optimum.

The approach is supported by the Pinery generator of structurally complex data.

References

[1] http://www.natcorp.ox.ac.uk/corpus/index.xml

[2] N. Oostdijk and P. Haan, Corpus-Based Research into Language. In honour of Jan Aarts, Amsterdam/Atlanta, GA, 1994, VII.

[3] M. L. Songini, QuickStudy: Extract, Transform and Load (ETL), 2004, Computerworld,

http://www.computerworld.com/s/article/89534/QuickStudy_ETL

[4] DTM Data Generator. http://www.sqledit.com/dg/

[5] Turbo Data. http://www.turbodata.ca/

[6] DBMonster. http://dbmonster.kernelpanic.pl/

[7] D. Chays, Y. Deng, P.G. Frankl, E.J. Weyuker, An AGENDA for testing relational database applications, Software testing, verification and reliability, 2004, VOL 14; PART 1, pages 17–44.

[8] C. Binnig, D. Kossmann, E. Lo, Testing database applications, 2006, Proceedings of the 25th ACM SIGMOD international conference on management of data / Principles of database systems, Chicago.

[9] A.V.Demakov, S.V.Zelenov, S.A.Zelenova. Pinery generator of structurally complex data: implementation of new capabilities of UniTESK // Proceedings of ISP RAS, Moscow, 2008, vol.14, part 1, 119–136.

[10] A.V.Demakov, S.V.Zelenov, S.A.Zelenova. Using abstract models for the generation of test data with a complex structure. Programming and Computer Software, 2008, vol.34, N 6, 341–350.

[11] I.B.Bourdonov, A.S.Kossatchev, V.V.Kuliamin, A.K.Petrenko. UniTesK Test Suite Architecture. Proc. FME'2002 conference, LNCS, 2391. Copenhagen, Denmark, 2002, 77—88.

[12] UniTESK Technology Web-site. http://www.unitesk.com/