DOI: 10.15439/2020F25 ISSN 2300-5963 ACSIS, Vol. 21

Data Quality Model-based Testing of Information Systems

Janis Bicevskis
Faculty of Computing
University of Latvia, Latvia
Email: Janis.Bicevskis@lu.lv
ORCID: 0000-0001- 5298-9859

Zane Bicevska
DIVI Grupa Ltd, Latvia
Email: Zane.Bicevska@di.lv
ORCID: 0000-0002-5252-7336

Anastasija Nikiforova
Faculty of Computing
University of Latvia, Latvia
Email: Anastasija.Nikiforova@lu.lv
ORCID: 0000-0002- 0532-3488

Ivo Oditis
DIVI Grupa Ltd, Latvia
Email: Ivo.Oditis@di.lv
ORCID: 0000-0003-2354-3780

Abstract—**This paper proposes a model-based testing approach by offering to use the data quality model (DQ-model) instead of the program's control flow graph as a testing model. The DQ-model contains definitions and conditions for data objects to consider the data object as correct. The study proposes to automatically generate a complete test set (CTS) using a DQ-model that allows all data quality conditions to be tested, resulting in a full coverage of DQ-model. In addition, the possibility to check the conformity of the data to be entered and already stored in the database is ensured. The proposed alternative approach changes the testing process: (1) CTS can be generated prior to software development; (2) CTS contains not only input data, but also database content required for complete testing of the system; (3) CTS generation from DQ-model provides values against which the system can be further tested. If the test results correspond to the values obtained during CTS generation, the system under test shall be considered to have been tested according to DQ-model. Otherwise, the user can verify the cause of the differences that may occur due incorrect software, as well as an inaccurate specification.**

Index Terms—**complete test set, data quality model, information system, model-based testing, symbolic execution.**

I. Introduction

SOFTWARE testing attracts the attention of researchers and practitioners since software development starts. Their main aim is to develop reliable software that can be used in the real-life circumstances. Unfortunately, this challenge has not yet been resolved and is far from being resolved. The proposed testing strategies and techniques are not able to ensure the reliability of software. Errors and bugs still cause system failures, despite millennial resources devoted to testing. According to Utting [1], software testing is a vital part of software development that requires between 30 and 60 percent of spent resources.

Model-based testing (MBT) is one of widely used solutions to improve the quality of the software. In scope of MBT, a model of information system (IS) is created according to which the system is tested. If IS works correctly on tests that cover all elements of the model, it is assumed that full/ complete system testing has been performed according to the selected model. For instance, if a program control graph is used as a test model, full/ complete testing is considered to have been performed if all the paths of the graph are executed. The advantages of MBT are also reflected in model-based testing user survey [2], according to which, respondents report on the average a 59% reduction in escaped bugs, 17% reduction in testing costs, and 25% reduction in testing duration.

The aim of this study is to propose an alternative model-based testing approach that uses a data quality model as a test model. As a result, a data quality (DQ) model-based testing approach called DQMBT is proposed. The DQ-model contains data objects and data quality conditions concepts where a data object describes real-world objects on which the information system accumulates data, while data quality conditions are aimed to describe the requirements that must meet the values of the attributes of data objects to be recognised as qualitative.

This paper is a continuation of [3], which addressed the basic concepts and introduced the overall structure of the proposed solution. According to [3], the main idea of the solution is as follows: as one of the main and primary tasks of the information systems is to collect and process data objects, the data to be entered must be tested first by verifying their correctness described by the conditions of the values of the data objects. The correct data objects can be stored in the database, while the information about the incorrect data objects must be provided to the data owner, allowing them to be edited and re-entered to the system. The verification of data objects must be carried out at two levels – syntactic and semantic/ contextual (in line with [4]). While syntactic control checks the relevance of the values of data objects attributes to the value syntax, semantic control checks the relevance of attribute values to the values of other data objects that have been already entered and stored in the database. The first use of the proposed solution is to compare the relevance of the data objects to be entered to the data objects already stored in the database, i.e. whether the data entered are correctly retained in the database. These checks must be described in the DQ-model and are not

The research leading to these results has received funding from the research project "Competence Centre of Information and Communication Technologies" of EU Structural funds, contract No. 1.2.1.1/18/A/003 signed between IT Competence Centre and Central Finance and Contracting Agency, Research No. 1.7 "The use of business process models for full functional testing of information systems"

related to implementation in the particular environment. Obviously, information systems are intended not only for collecting data but also for processing them, including, for calculating derived values, transformations etc. However, the primary task is to collect data, followed by many different tasks, so, this solution covers only one but nevertheless one of the main tasks of the information systems (in line with [5]). The second use of the proposed solution is to provide the complete testing capability of software that accumulates and stores data in the database. The values conditions for the attributes of data objects are proposed to be used to prepare test cases that will process all correct and incorrect cases. Using the DQ-model as a test model allows to prepare test cases constructively for the verification of all conditions. Testing software with these test cases, will check the accuracy of entering and storing data in both syntactic and contextual terms. The study therefore proposes a new complete testing criterion - verifying the correctness of all input data and its allocation in the database with tests that check all possible input values conditions.

This paper proposes not only the next set comprehensive set of concepts that are used to achieve the objective of the study being launched, but also provides an example demonstrating this idea, which has been promised in [3]. To sum up, the DQ-model based testing (DQMBT) approach for IS testing is proposed.

The paper deals with following issues: basic concepts and ideas addressed through related works (Section 2), the proposed solution (Section 3), analysis of the proposed solution (Section 4), conclusions (Section 5).

II. Related Works

This section briefly deals with the key concepts underpinning the proposed solution that are addressed through related works.

A. Testing basics

In software engineering, a test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path or to verify compliance with a specific requirement [6].

The modern definitions of testing underline that testing is a process aimed at verifying software compliance to requirements. An example of this is the definition provided by [7] in 2018, according to which, "*software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation. Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death*".

Many authors propose different and sometimes conflicting definitions of the concept of testing, in which, in some cases, the meaning of finding error and bug is exaggerated. As part of this study, the term "testing" should be understood in accordance with [8]: "*software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test*".

The viewpoint that testing aims to find bugs, errors and defects in software is outdated and no longer considered comprehensive and completely correct (in line with [7]). Methods that can find software bugs cannot be used to demonstrate that the software is working properly. In addition, despite numerous resources spent on testing, software almost always has bugs and errors.

To sum up, testing is a complex process, since tests are developed and accumulated throughout the whole software development process, starting with the development of a test for each individual function, ending with integration tests aimed at verifying the compatibility and integration of all components of the system.

B. Complete Test Set

Model-based testing opens up new horizons for software testing as it allows the creation of a test set for the selected and previously developed testing model that checks all the requirements for this model. Thus, the test model supports producing tests that fully cover aspects of the selected model. For instance, if a program control graph is used as a model, tests that execute all the graph arcs, are prepared. Such test set is called the complete test set (CTS). If the programme on this test set is working correctly, it shall be assumed that it has been sufficiently tested. Unfortunately, such a test criterion does not ensure the correct operation of the programme in all cases, but it is widely used as it allows for significant improvements in the overall quality of the software.

It is not a secret that theoretical studies on the possibility of automatically generating a complete test set according to a certain program code were carried out even in the 70s, when the first results on automatic generation of CTS were published in a cycle of articles on testing theory, including [9], followed by practical implementation [10]. During these studies, it was found that in cases where programming features are limited to processing a series of files, there is an algorithm capable of creating a complete test system for each such program. Thus, it can be assumed that for simple programmes that do not use complex language structures should also be possible to generate CTS.

This could complement unit testing with the possibility of testing with automatically generated test sets. Further studies demonstrate that if the program allows two two-way counters, the problem of constructing CTS is algorithmically unsolvable. This means that, depending on the programming languages, the impossibility of automatic generation of CTS soon occurs. In addition, despite the impressive age of this topic, it is still popular and widely used, as demonstrated by literature analysis [11]-[15]. In this study, the CTS corresponding to program control graph will be replaced by a DQ-complete test set system corresponding to the DQ-model addressed in the next Section.

C. Test Set Generation

According to the above section, if a program control graph coverage is used as a test model, testing according to this model means checking all possible control flow branching testing. As an example, this solution is provided by the Visual Studio 2015 Enterprise IntelliTest tool.

IntelliTest allows generation of unit test set for a particular class method or for all class methods simultaneously. For each condition, a test will be generated in the program code that will meet the specific condition when the method is operated. This tool analyses each condition branch in the C# code. The *if* branching conditions, statements and all operations that may constitute exceptions are analysed. As a result of the analysis, IntelliTest is designed to achieve the highest code coverage. In the generated test set, tests which have been performed, entry data and error message can be seen. The user may save them for further regression testing. The tool works with programs written in C# programming language. IntelliTest is based on a symbolic execution of a program that operates with symbolic notion of variable values instead of traditional command execution. This allows to establish the realizable conditions for paths, which, when resolved, result in input data for the execution of the corresponding path.

The concept of symbolic execution was introduced by Goodenough and Gerhart in 1975 [16], however, despite this, symbolic execution of programmes and specifications remain popular and become even more popular in recent years (see [17] – [21]). This study is not an exception, and symbolic execution is at the core of the proposed idea.

D. Data Quality Model

The study uses the previously proposed data object-driven data quality model (DQ-model) [4], consisting of 3 key components: (1) a data object defining the data to be analysed, (2) a specification of data quality, which defines the conditions to be met for the recognition of data as qualitative, and (3) a quality assessment process that determines the procedure that must be followed to assess the quality of data.

Each DQ-model component is represented by flowchart-like diagrams defined in a graphical domain specific language (DSL).

As in [22] the proposed solution will be demonstrated by some concrete almost classic example of university, more precisely, student and his achievements. Fig. 1 demonstrates the definition of the data object. Three data objects are defined: (1) *Students*, (2) its sub object *Course*, and (3) *inputMessage*, which contain data on specific courses passed, including course code, assessment, and date to be entered in the corresponding student list of grades (sub object *Course*) and stored in the database (data objects *Students* and *Courses*). Dashed lines represent contextual dependencies between the *inputMessage* attribute *studName* and *Students* instances.

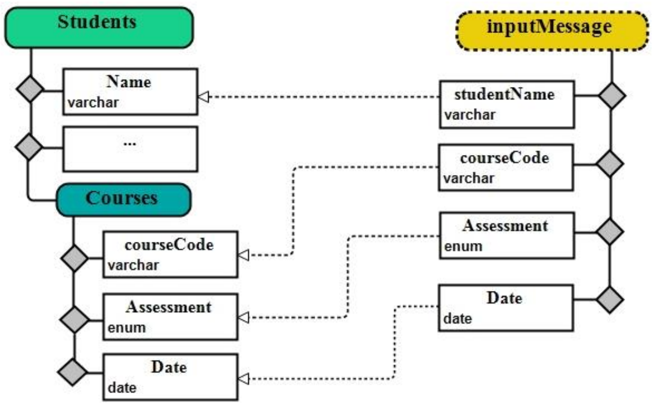


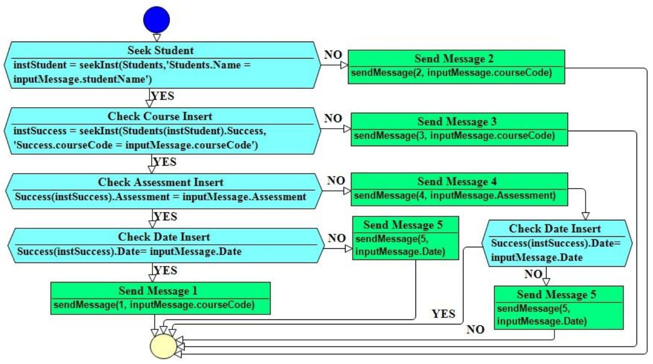
Fig.1 Data objects definition

Similarly, there is a contextual dependency between *inputMessage* attributes and *Courses* stored values. These dependencies are precisely defined in the quality conditions shown in Fig. 2, containing 4 checks:

- the *Students* data object has an instance, where ***Students.Name=inputMessage.studentName;***
- a new instance has been added to the *Students* sub-object *Courses*, where ***Courses.courseCode* = *inputMessage.courseCode***;
- a new instance with the corresponding course assessment has been added to the *Students* sub-object *Courses* data item, where ***Courses.Assessment* = *inputMessage.Assessment;***
- a new instance with the corresponding exam date has been added to the *Students* data object *Courses* sub-object, where ***Courses.Date = inputMessage.Date***.

Thus, a DQ-model used to generate tests is obtained from Fig. 1 and 2.

As stated in [4] and [23], the data object is defined according to the data to be analysed, so that parameters that are not relevant to specific users and use-cases are ignored (further denoted with "---" symbols). Data objects of the same structure form data object class. Similarly, the data quality specification shall also be determined by the user/ tester, depending on the use-case. The data quality specification can be defined informally or formally, but at the last stage all requirements are replaced by executable artefacts, such as SQL statements or program code, that further are executed.


Seek Student
instStudent = seekInst(Students,'Students.Name = inputMessage.studentName')
NO
Send Message 2
sendMessage(2, inputMessage.courseCode)
YES
Check Course Insert
instSuccess = seekInst(Students(instStudent).Success, 'Success.courseCode = inputMessage.courseCode')
NO
Send Message 3
sendMessage(3, inputMessage.courseCode)
YES
Check Assessment Insert
Success(instSuccess).Assessment = inputMessage.Assessment
NO
Send Message 4
sendMessage(4, inputMessage.Assessment)
YES
Check Date Insert
Success(instSuccess).Date= inputMessage.Date
NO
Send Message 5
sendMessage(5, inputMessage.Date)
Check Date Insert
Success(instSuccess).Date= inputMessage.Date
NO
Send Message 5
sendMessage(5, inputMessage.Date)
YES
Send Message 1
sendMessage(1, inputMessage.courseCode)
YES
NO
Fig.2 Requirements definition

The DQ-model is therefore executable. As proposed in [22] the DQ-model uses following methods to identify context of data objects:

- reviewing all class instances by changing the address ***<dataObjectName(instID).attributeName>***, calculated first by selecting the first instance using the ***instID = getFirst(dataObjectName)*** method, followed by the transition to the next instance using the method ***<instID = getNext(dataObjectName)>***. This option shall be used if the quality of the data is to be analysed for all instances of a particular data object;
- using a dynamically calculated address ***<instID = seekInst(dataObject, expression)>***, where an *expression* is a logical expression where operands are attribute names. If an instance of a data object is found as a result of an execution, (1) a reference to the data object is inserted into the variable *instID*, (2) the value *TRUE* is returned to the environment; otherwise, a *NULL* value is inserted in the variable that returns *FALSE*. This option is used if the quality is to be analysed for only one instance of a data object.

The effectiveness of the proposed approach has already been demonstrated by applying it to real data sets and presenting result in the series of articles.

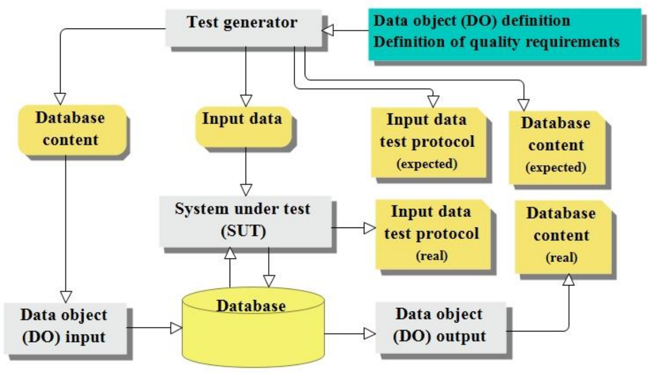
III. The Proposed Solution

This Section demonstrates how the proposed DQ-model can be used as a test model.

A. Data Quality Model as Testing Model

According to MBT principles, the test model is first selected. It serves to generate a set of tests that will test the correctness of the tested program or the system under test (SUT). The test set can be created either manually, partially or fully automatically. If the SUT on this test set works according to the specification, the SUT is considered to have been tested according to the selected model. As for criterion when SUT can be considered to tested sufficiently, a DQ-model coverage of all data quality requirements is selected. Although the proposed solution complies with the principles of the MBT, some important differences have to be mentioned: the proposed solution carries out a verification of the syntactical and contextual/ semantical control of input data and their correct allocation in data objects of the database. As it was mentioned in [3], it covers one of the most important tasks of the information systems, which is followed by other tasks such as calculations, reporting etc.

The proposed DQ-model-based test scheme or general architecture of DQMBT is shown in Fig. 3. The main actions are carried out by a "*Test generator*" using DQ-model to generate test input data, data object content (database) and two protocols – "*Input data test protocol (expected)*" and "*Database content (expected)*". The SUT is executed with generated test input data after the database content generated by the "*Test generator*" has been entered in the database.



Test generator
Data object (DO) definition
Definition of quality requirements
Database content
Input data
Input data test protocol (expected)
Database content (expected)
System under test (SUT)
Input data test protocol (real)
Database content (real)
Data object (DO) input
Database
Data object (DO) output

Fig. 3 Software verification procedure

The results of the SUT execution are recorded in the "*Input data test protocol (real)*" and the content of the data objects (database) are read after testing the SUT with generated test input data. The "*Input data test protocol (real)*" must coincide with the "*Input data test protocol (expected)*" generated by the "*Test generator*", although there are possible differences in formatting and texts. If these two protocols in general coincide with each other, it is assumed that the SUT is operating in accordance with the DQ-model, otherwise both protocols are sent to IS developers for further investigation of reasons of differences. Differences in protocols may indicate errors in the SUT or differences in the DQ-model from programmers' programs.

The proposed testing ensures complete testing according to the DQ-model, since all quality conditions are tested with generated test inputs and data object content, reaching their full coverage in both fulfilling and rejecting the conditions. In other words, complete/ full testing is performed according to the DQ-model. In addition, a specific test criterion is proposed, more precisely whether data to be entered is correctly allocated in data objects (database) without contradicting the data previously stored. It is clear, the proposed criterion does not guarantee the detection of all errors in the operation of SUT. For instance, SUT operation that record data in non-compliant locations in the database are not controlled, moreover, database integrity may be broken down. These types of errors cannot be detected even in the case of well-developed testing support tools.

The proposed approach is consistent with "black box" testing model because information on the internal design or implementation of the system is not used. Only the DQ-model is used to generate tests. However, it should be acknowledged, that the SUT may contain activities that are not covered by the DQ-model and the tests generated therefore cover the operation of SUT only partially (this is a common challenge for MBT).

This means that either traditional testing methods should be used, or the testing model should be enriched with new features.

The next section addresses the test generation algorithm.

B. Algorithm of Test Generation

In the proposed algorithm, the first phase requires the deployment of a requirements/ condition model (Fig.2) into a tree-like chart given in Fig. 4. The last branch vertices contain numbers, which in the scope of the provided example range from 1 to 6. The tree contains only one node with a number 1 that represents correct data processing from syntactic and semantic/ context checks to correct data allocation and storage in the database. The branch with number 2 represents a violation of the input data context since database does not have data for the specific student. Branches with numbers from 3 to 6 indicate incorrect data allocation in database.

In the second phase of the algorithm, the conditions for the realisation of the corresponding branches are established using the symbolic execution of the DQ-model conditions. For instance, the conditions for the first branch under this example are:

- **exist Students(instStudent)** where ***inputMessage.studName=Students(instStudent).name***
- **exist Course(instCourse)** where ***inputMessage.courseCode=Course(instCourse).name***
- **valid Assessment** where ***inputMessage.Assessment = Course(instCourse).Assessment***
- **valid Date** where ***inputMessage.Date = Course(instCourse).Date.***

When resolving the conditions for the branch realisation in all 6 cases, the test input data is obtained shown in Table I and the content of the data objects (database) in Table II and III. Each instance of a data objects serves as test input data to complete one of the 6 branches. The 6 rows of the Table I correspond to 6 branches that when executed fulfil all the data quality conditions transitions/ paths of the DQ-model.

In other words, the generated test set is a full/ complete DQ-test set. Execution of the SUT with all 6 tests will achieve the complete testing of the SUT according to the DQ-model criterion.

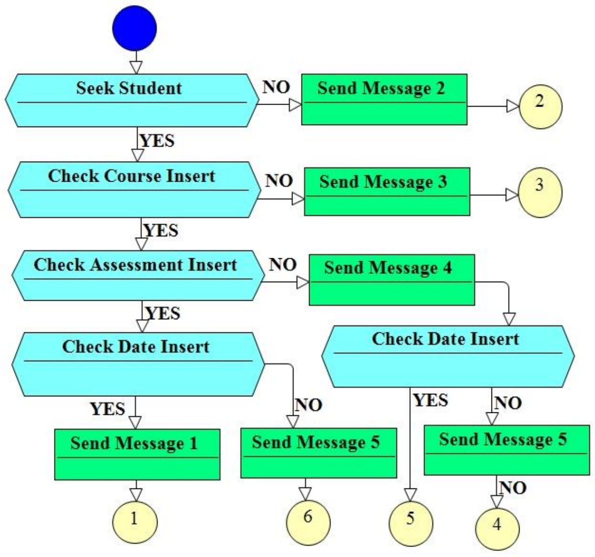


Fig.4 Requirements tree

TABLE I.
DATA OBJECT INPUTMESSAGE

studName	courseCode	Assessment	Date
stud-1	course-1	assess-1	date-1
stud-2	---	---	---
stud-3	course-3	---	---
stud-4	course-4	assess-4	date-4
stud-5	course-5	assess-5	date-5
stud-6	course-6	assess-6	date-6

TABLE III.
DATA OBJECT STUDENTS

Name	Address
stud-1	---
stud-3	---
stud-4	---
stud-5	---
stud-6	---

TABLE IIIII.
DATA OBJECT COURSES

courseName	Assessment	Date
course-1	assess-1	date-1
course-4	assess-4	date-4
course-5	assess-5	date-5
course-6	assess-6	date-6

The test theory generally understands the concept of "test" as the analysis of the value of the data to be entered and the expected results of the SUT execution obtained by the SUT with the entered data. It is known that the result of execution depends not only on the data to be entered but also on the content of the related data objects (database). Thus, the proposed approach generates not only the data to be entered but also the database content that ensures the execution/ completion of the chosen path. This can be achieved by symbolically executing the contextual conditions/ requirements between the interrelated data object of the DQ-model (as shown in Fig. 2).

The next stage of the algorithm supposes the execution of the conditions with the DQ-complete test set (input entered and the generated content of the data objects) that results in obtaining the expected test results, so called benchmark.

When testing the SUT with the previously generated DQ-complete test set, the results of execution must [by its nature] coincide with the results of execution of the DQ-model or benchmarks. Thus, it can be argued that the test objective has been achieved since the tested programme is tested with input data that ensures verification of all data quality conditions, as well as checking the compliance of input data with their retention on the database.

C. Testing Process

At the next stage, the SUT is tested with automatically generated tests. The values of generated data objects are sent to the database that is done by separate procedure (individual for a particular system). This is followed by SUT testing with the DQ tests given in Table I – "*inputMessage*".

TABLE IV.
PROTOCOL

branch	message#	text
1	1	**Message-1**: input successful: *<stud-1, course-1, assess-1, date-1>*
2	2	**Message-2**: **input error: invalid StudName** *<stud-2, course-2, assess-2, date-2>*
3	3	**Message-3**: **database error: invalid courseCode** *<stud-3, course-3, assess-3, date-3>*
4	4, 5	**Message-4**: **database error: invalid Assessment** *<stud-5, course-5, assess-5, date-5>* **Message-5**: **database error: invalid Date** *<stud-5, course-5, assess -5, date-5>*
5	4	**Message-4**: **database error: invalid Assessment** *<stud-5, course-5, assess-5, date-5>*
6	5	**Message-5**: **database error: invalid Date** *<stud-5, course-5, assess-5, date-5>*

The results of the SUT execution must be consistent with the previously obtained protocols. In the case of differences, the inconsistencies between the operation of the SUT and the DQ-model are identified (Table IV).

This can be caused by both errors in the SUT or errors in the specification – the DQ-model. To automate the testing process, most test support tools support the SUT execution with a user-selected set of tests [24] – [27]. These tests usually are accumulated gradually using the same test support tools. As a result, in most cases the tests records formats are internal formats of these tools which are not related to the tested programs. In the proposed case, the situation is more complicated because all generated tests must be able to be performed automatically in one session. This can be achieved by preparing test drivers that establish database content, calls the cyclic execution of the SUT with all generated tests, and read the database content after completion of the tests (see Fig. 5).

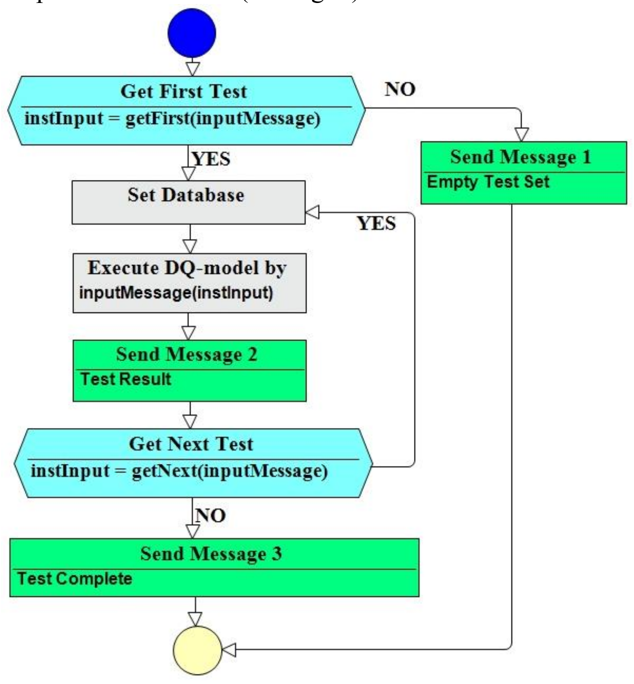


Fig.5. Testing process

IV. ANALYSIS OF THE PROPOSED SOLUTION

The traditional and common approach to software testing is to define and plan test cases prior to their execution and then compare their results with documented expected results [28]. The proposed approach differs from such test process when the tester prepares test cases based on an informal specification, his own experience or intuition without an exact and precise specification of the operation of the SUT.

The proposed approach uses a formal and at the same time executable specification, generates the DQ-complete test set and the expected results of its execution or benchmarks. After automated testing of the SUT, the tester should only compare the results obtained with the expected benchmarks.

The tester does not have to prepare the test by himself and perform the execution of the SUT with them. The quality of testing therefore does not depend on the qualification of the tester, but on the quality of the DQ-model in the way of its accuracy and completeness, i.e. whether the testing model meets the requirements of the system.

From the beginning of testing, it is well known that systematic testing reveals most errors, and after each iteration, the number of errors at the users' end is lower. Even at the end of the 20th century, it was already known that, when tests are selected intuitively, the end user receive 8-10 times more errors compared to when tests are selected based on the formalized model [29].

This time, systematic testing is understood as testing according to the MBT principles. The main advantages of using a complete test set (CTS) in the testing process are:

- a complete test set (CTS) can be generated prior to the development of the programme, thus, it can serve as an additional interpretative example of the specification;
- SUT testing may be initiated immediately after the development of the programme;
- the CTS shall completely verify the syntactic and semantic/ context requirements specified in the specification;
- the tester shall be released from the development of tests and their execution.

However, together with the advantages, some of the challenges and limitations of the proposed solution should also be mentioned:

- the proposed solution supposes the development of a DQ-model that requires resources and specific expertise and knowledge in the development of DQ-models although their development is not too complicated, especially for people with IT background;
- additional tools such as test generator, database content input, output of results, CTS execution driver, ensuring cyclic execution of all CTS tests without the involvement of the tester, are required to support testing;
- the SUT must be prepared for its automated testing with CTS.

V. Conclusion

The solutions proposed by the test theory are only partly capable of meeting the practice needs, since the proposed testing strategies and techniques do not guarantee the development of qualitative programmes. The previous testing paradigm as a search for errors and bugs is now switching to the new one, according to which, testing is tasked with achieving reliable software. This can be achieved through systematic testing, for instance, using model-based testing.

The study therefore proposes an alternative, model-based testing approach called DQMBT, based on a data quality model proposed in previous studies. It defines the data objects and conditions that must meet the parameter values of the data objects to consider the data object to be correct and qualitative. The proposed test algorithm provides such two main features as:

- the generation of a DQ-complete test set to check the correctness of the operation of the programs to be tested, covering all possible quality conditions for the input data;
- the comparison of the relevance of the data objects to be entered and stored in the database to one another, verifying whether the data entered is correctly stored in the database.

The program testing with an automatically generated complete test set (CTS) changes the testing process significantly as the preparation and execution of individual test cases is replaced by complete testing that systematically checks all conditions in a single session.

Using a data quality model as a test model does not solve all program testing problems. The proposed approach covers only a part, however, a very important part of the functional testing of the information systems, more precisely complete testing of the input data and testing of the relevance of data stored in the database with input data. This would lead to a significant improvement in the overall quality of information systems, which today is one of the most important challenges we need to solve [30].

In addition to an in-depth study on the concepts, which are not thoroughly covered in this paper mentioned in Section 4, further studies on the topic include the application of the proposed approach to the real system we are dealing with. This will not only allow a test of the proposed approach, but also lead to a quantitative and qualitative results that could be compared with other strategies currently in use. Then, the question on how to handle system events, when one event takes place more quickly than others, but affects the result of previous events, will be addressed.

References

[1] M. Utting, B. Legeard. Practical model-based testing: a tools approach. *Elsevier*, 2010.

[2] R.V. Binder. 2011 Model-based Testing User Survey: Results and Analysis. *System Verification Associates. System Verification Associates*, 2012.

[3] A. Nikiforova, J. Bicevskis. Towards a Business Process Model-based Testing of Information Systems Functionality. In *Proceedings of the 22nd International Conference on Enterprise Information Systems* - Volume 2: *ICEIS*, ISBN 978-989-758-423-7, pp. 322-329, 2020. DOI: 10.5220/0009459703220329.

[4] A. Nikiforova, J. Bicevskis, Z. Bicevska, I. Oditis. User-Oriented Approach to Data Quality Evaluation. *Journal of Universal Computer Science, 26(1)*, pp.107-126, 2020.

[5] R. Perez-Castillo, A.G. Carretero, M. Rodriguez, I. Caballero, M. Piattini, A. Mate et al. Data quality best practices in IoT environments. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC),* pp. 272-275. IEEE, 2018, DOI: 10.1109/QUATIC.2018.00048.

[6] 24765-2010 - ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary, doi:10.1109/IEEESTD.2010.5733835. ISBN 978-0-7381-6205-8.

[7] K. Olsen, T. Parveen, R. Black, D. Friedenberg, E. Zakaria, M. Hamburg, J. McKay, M. Walsh, M. Posthuma, M. Smith, R. Smilgin, S. Ulrich, S. Toms. Certified tester foundation level syllabus. *Journal of International Software Testing Qualifications Board*, 2018.

[8] P. Saini. Revisiting Mutation Testing in Cloud Environment (Prospects and Problems), *A Journal of Composition Theory*, Volume 12, Issue 9, pp. 2007-2011, 2019, DOI:19.18001.AJCT.2019.V12I9.19.10519.

[9] J. Bārzdiņš, J. Bičevskis, A. Kalniņš. Automatic construction of complete sample systems for correctness testing. *Math. Found. of Computer Science. Springer Verlag, Berlin*, 1975.

[10] J. Bičevskis, J. Borzovs, U. Straujums, A. Zariņš, E.F.jr. Miller. SMOTL - A System to Construct Samples for Data Processing Program Debugging. *IEEE Transactions on Software Engineering*, Vol. SE-5, No.1, pp. 60-66, 1979.

[11] Y. Y. Lin, N. Tzevelekos. Symbolic Execution Game Semantics. *arXiv preprint arXiv:2002.09115*, 2020.

[12] G. P. Farina, S. Chong, M. Gaboardi, M. Relational symbolic execution. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, pp. 1-14, ACM, https://doi.org/10.1145/3354166.3354175.

[13] M. Aggarwal, S. Sabharwal. Combinatorial Test Set Prioritization Using Data Flow Techniques. *Arabian Journal for Science and Engineering, 43(2),* pp. 483-497, 2018, https://doi.org/10.1007/s13369-017-2631-y.

[14] M. Handique, J. K. Deka, S. Biswas, K. Dutta. Minimal test set generation for input stuck-at and bridging faults in reversible circuits. In *TENCON 2017 IEEE Region 10 Conference*, pp. 234-239, DOI: 10.1109/TENCON.2017.8227868.

[15] G. Eleftherakis, P. Kefalas, E. Kehris. A methodology for developing component-based agent systems focusing on component quality. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 561-568. IEEE, 2011.

[16] J. Goodenough, S. Gerhart, S. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, Vol. 1 (2), pp. 156-173, 1975.

[17] A. Ibing. Efficient data-race detection with dynamic symbolic execution. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS),* pp. 1719-1726. IEEE, 2016, DOI: 10.15439/2016F117.

[18] J. Bicevskis, G. Karnitis. Testing of Execution of Concurrent Processes. *Proceedings of DB&IS'2020* (to be published).

[19] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3), pp. 1-39, 2018, https://doi.org/10.1145/3182657.

[20] D. Trabish, A. Mattavelli, N. Rinetzky, C. Cadar. Chopped symbolic execution. *In Proceedings of the 40th International Conference on Software Engineering*, pp. 350-360, 2018, https://doi.org/10.1145/3180155.3180251.

[21] R. Stoenescu, M. Popovici, L. Negreanu, C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 314-327, 2016, https://doi.org/10.1145/2934872.2934881.

[22] J. Bicevskis, A. Nikiforova, Z. Bicevska, I. Oditis, G. Karnitis. A Step Towards a Data Quality Theory. In *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pp. 303-308. IEEE, 2019, 10.1109/SNAMS.2019.8931867.

[23] J. Bicevskis, Z. Bicevska, A. Nikiforova, I. Oditis. Towards Data Quality Runtime Verification. In *2019 Federated Conference on Computer Science and Information Systems (FedCSIS),* pp. 639-643. IEEE, 2019, DOI: 10.15439/2019F168.

[24] V. Garousi, F. Elberzhager. Test automation: not just for test execution. *IEEE Software,* 34(2), pp. 90-96, 2017, DOI:10.1109/MS.2017.34.

[25] D. M. Rafi, K. R. K. Moses, K. Petersen, M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST),* pp. 36-42. IEEE, 2012.

[26] P. Loyola, M. Staats, I.Y. Ko, G. Rothermel. Dodona: automated oracle data set selection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis,* pp. 193-203, 2014, http://dx.doi.org/10.1145/2610384.2610408.

[27] H. Kaur, G. Gupta. Comparative study of automated testing tools: selenium, quick test professional and testcomplete. *Int. Journal of Engineering Research and Applications,* 3(5), pp. 1739-1743, 2013.

[28] W. Afzal, A. N. Ghazi, J. Itkonen, R. Torkar, A. Andrews, K. Bhatti, K. An experiment on the effectiveness and efficiency of exploratory testing. *Empirical Software Engineering,* 20(3), pp. 844-878, 2015, https://doi.org/10.1007/s10664-014-9301-4.

[29] J. Bicevskis. The Effectiveness of Testing Models. In Proc. of 3rd Intern. *Baltic Workshop "Databases and Information Systems",* 1998.

[30] E. Ziemba, T. Papaj, D. Descours, Assessing the quality of e-government portals-the Polish experience. In *2014 Federated Conference on Computer Science and Information Systems,* IEEE, 2014, pp. 1259-1267, http://dx.doi.org/10.15439/2014F121.