Data Quality Model-based Testing of Information Systems: the Use-case of E-scooters

Anastasija Nikiforova
Faculty of Computing
University of Latvia
Riga, Latvia
Anastasija.Nikiforova@lu.lv

Janis Bicevskis
Faculty of Computing
University of Latvia
Riga, Latvia
Janis.Bicevskis@lu.lv

Zane Bicevska
DIVI Grupa Ltd
Riga, Latvia
Zane.Bicevska@di.lv

Ivo Oditis
DIVI Grupa Ltd
Riga, Latvia
Ivo.Oditis@di.lv

***Abstract*- The paper proposes a data quality model-based testing methodology aimed at improving testing methodology of information systems (IS) using previously proposed data quality model. The solution supposes creation of a description of the data to be processed by IS and the data quality requirements used for the development of the tests, followed by performing an automated test of the system on the generated tests verifying the correctness of data to be entered and stored in the database. The generation of tests for all possible data quality conditions creates a complete set of tests that verify the operation of the IS under all possible data quality conditions. The proposed solution is demonstrated by the real example of the system dealing with e-scooters. Although the proposed solution is demonstrated by applying it to the system that is already in use, it can also be used when developing a new system.**

Keywords— complete test set, data quality model, e-scooters, Internet of Things, IoT, Internet of Vehicles, model-based testing, symbolic execution.

I. Introduction

Software correctness is an issue that has been debated since the beginning of programming. Although at the beginning software testing meant a testing together with debugging, testing as a separate and independent step appeared only in the 70s. Nowadays, the main goal of most studies on software testing is to provide reliable software that could be used in everyday life, but, this goal is not succeeded, yet. Despite numerous resources are spent on testing, errors and bugs in software still causing system failures. In addition, software testing is often done manually because its automation appears to be sufficiently complex and as a result, the level of effectiveness of such a testing is low [1], [2].

The study launched aims to propose a new technology for testing and software development that would take a step towards the main objective of developing technology for reliable software development. The study addresses related topic by covering just one specific part of an information systems (IS) that nevertheless is one of the main tasks – the correctness of input messages which are inserted and their correct allocation in database. This part is very vital since this is followed by various different tasks which depend on the stored data. The main idea of the approach is: (a) first, the description of the data to be processed by IS and its processing rules are developed; they are further used to develop a complete set of tests, (b) then, an automated test of the system on the generated tests is performed verifying the operation of the IS under all possible data quality conditions [3]. This allows to complement traditional software testing with automated checks of compliance of data entered as a result of automated business processes and stored in a database. This aimed to improve the complete testing methodology of IS using business process and data quality models. The proposed solution can be applied to both, software that is being developed and already in use.

The idea of the proposed solution is close enough to the philosophy of model-based testing (MBT) - a testing model that is further used to prepare tests is created. When testing the program on these tests, the correct operation of the program on these tests shall be achieved. One of the main tasks in MBT is to find a model that would express the behaviour of the program in detail. The previous document on this study [3] suggested to use a “black box” testing method based on a test of all requirements put forward the program. Since in the case where the requirements are defined in the natural language, misunderstandings are possible, it is important to define these requirements very precisely, which can be achieved through the use of domain specific languages (DSLs). In addition, the use of DSL makes it possible to undertake full/ complete testing. Thus, the previously proposed data object-driven data quality model, defined using graphical DSLs [4], has been adapted to this study.

While the first steps towards this idea were already proposed in [3] covering key concepts of the solution to be developed, motivating their choice among a wide range of alternatives, this paper is a next step towards an idea, proposing more detailed and clarified vision of the key elements, detailed architecture of the proposed solution, demonstrating all the concept on a real-life example. This paper aims to provide an algorithm that would allow a complete set of tests from the system model to be tested, more precisely, part of the system, and to check the operation of the system under these tests. The topicality of this study is demonstrated through addressing very specific example – electric scooters (e-scooters).

Electric scooters, which popularity continuously increases in recent years all around the world, are an example of Internet of Things (IoT) devices or even more specifically Internet of Vehicles that allows demonstrating wide possibilities of the proposed solution. The proposed idea focusing mainly on data, become even more clear when we speak about e-scooters which together with their riders’ smartphones continuously generating data from integrated sensors, which are transmitted to the systems of the companies that own them. Data regarding the location of each connected vehicle, how long each ride takes, which docks need to be restocked, and which ones are full are always available in real time. However, it is also known from our own experience, and from the news and studies that this invention brings a lot of challenges that need to be addressed and solved. This study cover issues related to data obtained from e-scooters [and smartphones], which may be inaccurate and noisy.

The paper addresses the following issues: a review of related studies (Section 2), a description of the solution (Section 3), a use-case of e-scooters (Section 4), and conclusions (Section 5).

II. RELATED WORKS

A. Researches on Data Quality

The data quality topic is often addressed separately from testing. There are known studies covering and extending ISO quality standards, such as adapting it to IoT, sensor data, etc. and provides a detailed a state-of-the-art analysis [5]-[8], therefore, this study will not repeat this discussion. In addition, our previous studies covered the topic of data quality without linking it to the testing of information systems as well.

This time it was decided to propose a testing approach based on a previously proposed data quality model [9]-[11] called DQ-model-based testing (DQMBT). As follows from the title of the proposed testing approach, it belongs to model-based testing methods. This topic was briefly discussed in [3] and will not be addressed here, focusing on another most specific topic, more precisely symbolic execution and on existing studies on symbolic model-based testing, which underpins the proposed approach.

B. Symbolic Execution and Symbolic Model-based testing

While the symbolic execution was invented in the late 70s [12], it remains popular and widely used. This study is not an exception, so let us briefly discuss the concept of symbolic execution and symbolic model-based testing. It is a well-known fact that system can have an infinite number of possible states, input data, possible behaviours etc. Therefore, some studies decide to cover some part, hoping that it will be enough, however, this is a very risky choice. Therefore, another option is to find possibilities to cover set of infinite input values, states etc. Here comes **symbolic execution** which reduces the number of possible paths by associating classes of inputs.

According to [13], the first goal of symbolic execution is to explore the possible execution paths of an application. The difference between symbolic execution and informal testing with sample input values is that the inputs in symbolic execution are symbols representing classes of values. For example, if a numeric value is expected by the application, a generic representing the whole set of numerical values is passed. Obviously, the output of the execution will be produced as a function of the introduced input symbols. Solving path condition equations is crucial both for the symbolic execution itself and for test case generation. During symbolic execution it is be necessary to constantly evaluate the **path condition equations** in order to decide whether the control path being explored is feasible or not. In the case (1) if there is no solution to the equations at some moment, the path is infeasible, however, (2) for each **reachable/ feasible control path** the path condition provides the relation between input variables that will direct execution through that particular path - test case generation occurs. If it is possible to generate values that satisfy that relation, then it is possible to extract a test case. By finding solutions to the equations that describe control paths it is possible to extract values that can be used as testcases. These values will clearly force the application to follow the control path that defines that path condition (also in line with [13]).

Test generation using symbolic execution is commonly divided into two groups – **static** and **dynamic**. Static test generation consists of analysing a program statically, by using symbolic execution techniques to attempt to compute inputs to drive particular program along specific execution paths or branches, without ever executing the program. [14] states that static test generation is doomed to perform poorly whenever precise symbolic execution is not possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision. At the same time, dynamic test generation consists of executing the program, typically starting with some random inputs, while performing symbolic execution dynamically, collecting symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. This process is repeated until a given final statement is reached or a specific program path is executed. Dynamic test generation can be viewed as extending static test generation with additional runtime information, and is therefore more general, precise, and powerful. However, it is obvious that it is more resource consuming compared with the static execution. The list of examples of both, dynamic and static test case generators are available in [14]. Our study uses static test case generation as it satisfies the aim of the proposal.

Now, let us briefly cover some studies on symbolic model-based testing. For instance, [15] presents a technique for the automatic generation of real-time black-box conformance tests for non-deterministic systems, where timed automata with a dense time interpretation cannot be analysed by finite state techniques, since the timed transition system associated with it has infinitely many states. Therefore, authors gave their preference to symbolic execution, in order to analyse the specification, to synthesize the timed tests, and to guarantee coverage with respect to a coverage criterion that are found by authors as the most efficient way dealing their issue. They present an algorithm and data structure for systematically generating timed *Hennessy* tests that ensures that the specification will be covered such that the relevant Hennessy tests for each reachable equivalence class will be generated. To compute and cover the reachable equivalence classes, and to compute the timed test sequences, they employ symbolic reachability techniques based on constraint solving (adapted for model checking of timed automata). One of the specific notes should be mentioned here, that in addition to other known advantages of symbolic execution, authors mention that symbolic execution is much less sensitive to the clock constants and the number of clocks appearing in the specification compared to the region construct. As a result, a prototype was developed proving by an example that resulting test suite is quite small (finite), and is constructed quickly, and with a reasonable memory usage demonstrating advantages of symbolic execution.

According to [13], there are different ways for test case generation, more precisely, abstract **model-based test case generation** and **code-based test case generation**. As for abstract model-based test case generation, these studies start from an abstract specification and perform searches through the execution state space of the specified application by using a constraint logic programming language [13]. This search is done in a symbolic fashion in the sense that each state of the model corresponds not to a single concrete state but rather to a set of constrained model input variables. The constraints for the model input variables at a given state are calculated by symbolically executing the path until that state. Here, [16] can be mentioned as an example applying this idea to a specific case of smart cards and boundary testing idea [17]. As for code-based test case generation, as it follows from the name of this approach, test cases are generated directly from a real code. In this case, the model does not exist explicitly and is provided implicitly with the temporal logic formulas. The expected correct and incorrect behaviours of the implementation are described by the test engineer using temporal logic. The simple fact that the witnesses or counter examples to these formulas exist already provides information about the correctness of the implementation. This approach is less popular due the fact it has a list of disadvantages (see [13]).

There are also known studies proposing enabling symbolic test generation for input/ output automata [18]. [19] proposes a complete formal framework for symbolic testing which aims to overcome a list of challenges, including the loose of structure and information available in the data definitions and constraints, and to avoid infamous state space explosion problem, which limits the usability of test generation tools that are met when the symbolism does not take place. Authors underline that the introduction of symbolism avoids the state-space explosion during test generation, and it preserves the information present in data definitions and constraints for use during the test selection process. As a result, *sioco* - fully symbolic version of *ioco* (input/ output conformance) is proposed that turned out to be an improved version of the initial version (*ioco*) because of symbolism.

To sum up, it is concluded that symbolic execution can be used with both, abstract and concrete models, and it is found to be suited for the study presented since it allows to perform complete testing of the particular part of the system.

III. Data Quality model-based Testing

A. *Data Quality Model as a Testing Model*

The proposed solution is based on the use of data object-driven data quality model previously proposed in [4], [9]-[11]. This model proved to be simple and effective enough that was proved by a cycle of studies (see [11]). Its effectiveness was demonstrated by applying it on the real open data identifying their data defects, despite the different structures of data sets and the complexity of data. Let us briefly cover main components that together constitute the data quality model, emphasizing key points in scope of the given study:

1) **data object** - a set of attribute values that characterize one real object that defines the data to be analysed. A data object can be ***primary***, ***secondary,*** and ***sub-object***. A collection of data objects of the same structure forms a **data object class**.

   The creation of data object in this study is performed twice: (1) for data correctness analysis, by performing mainly syntactic check in the scope of one data object, (2) for data correct allocation in the DB, by analysing this in scope of multiple data objects extracted from the database, where data object received at the first step is considered to be a primary, but those, which are extracted from the database – secondary data objects, where both primary and secondary data objects can have an arbitrary number of sub-objects.

   As in [4] the address for the attribute value of a single data object is <*dataObjectName.attributeName*>. This address is used at the stage of determining data quality requirements;

2) **data requirements/ conditions** that determine the conditions that must be met to admit the data as qualitative. Requirements regarding attributes of a data object are used to prepare/ generate test cases, which would cover all correct and incorrect inputs.

Both components are represented by their own graphical Domain Specific Language (DSL). Since they have proved its applicability to these tasks, they are refined in the case of this study (in line with [20]). In scope of this study this DQ- model is even more suited since these tasks are supposed to be performed by IT-specialists who will gain profit of a graphical data quality model but also reducing the risk of incorrect models which previously should be done by “clever users”. In addition, data quality model is not related to the information system that has accumulated data.

According to [3], the database (DB) is checked before and after data insertion. Before inserting data, the DB should be checked to make sure that a particular insert is possible. At this stage, only *read* operation is performed. After data entry, the data object corresponding to the DB is read and compared with the input data. In case of differences, an invalid data entry is identified. Thus, data entry into the DB is controlled by an external procedure, that, after receiving the message that needs to be entered, checks the preconditions for execution before data is entered into the database and checks whether this has happened correctly in the database after data entry.

When data objects and conditions for input message, the data objects retrieved from the database are defined and the conditions that apply to these data objects are applied, a test generation step is taken. However, the test generation is only possible after selection of test selection criteria. Different testing models and their coverage can be selected as a criterion. Since we are mainly discussing one specific testing objective, we limit the remaining components to this specific purpose and involve very specific criteria – (1) to verify whether the data to be entered is correct and (2) whether they have been correctly allocated in the database without contradictions to internal constraints. Thus, the criterion when system under test (SUT) can be considered to be sufficiently tested is the full coverage of defined data quality requirements/ conditions.

B. Algorithm

The proposed algorithm uses a data quality model (DQ-model) as a testing model and is able to generate a DQ-complete test set (CTS). Its main steps are:

- **step 0:** creating DQ-model covering both, input messages and data retrieved from the database (according to previous subsection);
- **step I**: data quality conditions defined in flowcharts are expanded in a tree-like format;
- **step II**: for each tree branch, the condition for its feasibility/ reachability shall be established by means of symbolic execution;
- **step III:** resolving the conditions for branch reachability results in tests, containing both, input data values, data objects (database) content, branch execution results. The obtained test set is a **DQ-complete test set** that ensures testing of all conditions and the results to be stored in the database.

The general architecture of the proposed solution is shown in Fig. 2. As was already stated in [21], the main actions are carried out by a "*Test generator*" using DQ-model to generate test "*Input data*", "*Database content*" (data objects for data retrieved from DB) and two protocols – (1) "*Input data test protocol (expected)*" and (2) "*Database content (expected)*".

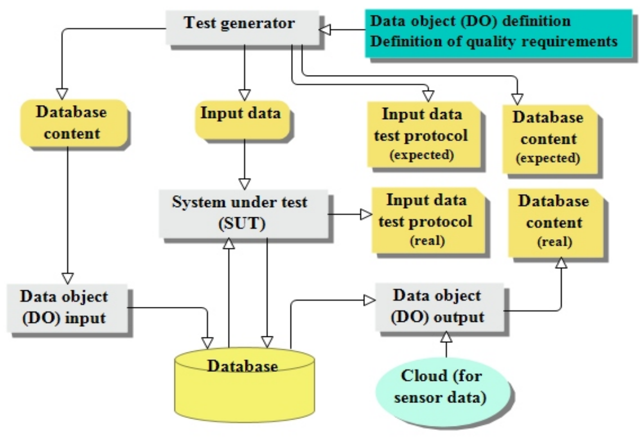

Fig. 1 Software verification procedure.

The SUT is executed with generated test *Input data* after the "*Database content"* generated by the "*Test generator*" has been entered in the database. 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. Expected values are considered **benchmarks** with which real values are compared. If these two protocols coincide with each other, it is assumed that the SUT is operating according to the DQ-model, otherwise both protocols are sent to IS developers for further investigation of reasons of differences, which may indicate (a) errors in the SUT or (b) differences in the DQ-model from programmers' programs. In the first case, the error is detected and must be corrected, in the second case, the model must be re-checked and improved to eliminate the differences found.

As for a test suite, the given study supposes symbolic test suite that is execution of data quality requirements (its suitability for complete testing is discussed in [22]). For each case of the requirements, the conditions of the requirements are established which, when resolved, result in specific tests which further test the system.

The positive point of this choice is that the finite number of tests can be performed to cover infinite set of possible cases. A test suite is considered to be complete with respect to a specific fault model, if a system under test whose behaviour conforms to the reference model passes all test cases (called *soundness*), and every non-conforming system under test fail at least one test case (called *exhaustiveness*) [23]. It is clear that to ensure soundness of the produced tests, symbolic reachability analysis is needed to select only states for testing that are reachable, and to compute only timed traces that are actually part of the specification (in line with [13]).

To sum up, the proposed approach uses a formal executable specification, which generates the DQ-complete test set and the expected results of its execution or benchmarks. It means, that when automated testing of the SUT is performed, the tester should only compare the results obtained with the expected benchmarks. A more detailed overview of these steps is demonstrated in the next Section on the real-life example.

IV. DQMBT Approbation on the E-scooter System

The solution proposed is applied to one particular local system of e-scooters we deal with. E-scooters are becoming more and more popular for a variety of purposes that are not limited to individuals but also for other purposes, such as supply of goods [24], [25] worldwide and also in Latvia. However, with significant benefits, such as the absence of emission, silence, relatively small sizes, they are also characterised by some limitations. There are many topics addressing issues related to the deficiencies and use of e-scooters, starting with their unsafety, traffic offences etc. In addition, some authors argue for data quality issues observed in the collected data such as [26], where the error rate of data, in the sample of 300 records, is 3%. This is not a surprise, especially given that in the case of e-scooters, data are collected from a number of sources – users smartphones, sensors, which are then transmitted in a system for which a proposed DQMBT approach can been applied. The system under test in the scope of this study is not an exception. The nature of e-scooters requires the collection and storage of every usage and user data. The challenge here is the huge amount of data need to be processed, however, the advantage is that data collected in the most cases is structured or semi-structured.

Two the most well-known problems that were identified by our own (for the system we deal with) are (1) inaccurate data on the charge of scooters which may exceed 100% or may change over few second by such a high number of percent that cannot be true (from 80% to 20% and then to 78%), (2) inaccuracies in data received from sensors, more precisely, according to these data, in one time interval e-scooter may be transported in thousands kilometres from its current position that cannot be true. However, *whether these issues are single*? *what if there are many other issues have not noticed, yet*? It is clear that it is not enough to base conclusions on the IS system on observations only and mechanism should be involved controlling this. While the solution is a new technology for testing and software development, this example covers the case when already developed system is under test, however, it can be used while developing a new system as well. Let us demonstrate the proposed algorithm step by step.

A. Step 0: Creating DQ-model

According to the data quality model used, the first step is to create data objects (Fig. 2) that are (1) *Scooters* with its sub-object *Ride*, (2) *Customers* with its sub-objects (2a) *Cards* and (2b) *Payments*, (3) 3 input messages – (3a) "*Scooter Heartbeat*", (3b) "*Ride:Start*" and (3c) "*Ride:Finish*" received from a scooter at specific stages of the ride, i.e, when the ride begins, during the ride and when it completes, affecting the nature of data obtained from the scooter and its allocation in the database. In real circumstances, the proposed approach is supposed to process data simultaneously from both a smartphone and an e-scooter (only data from e-scooter is under analysis at this time).

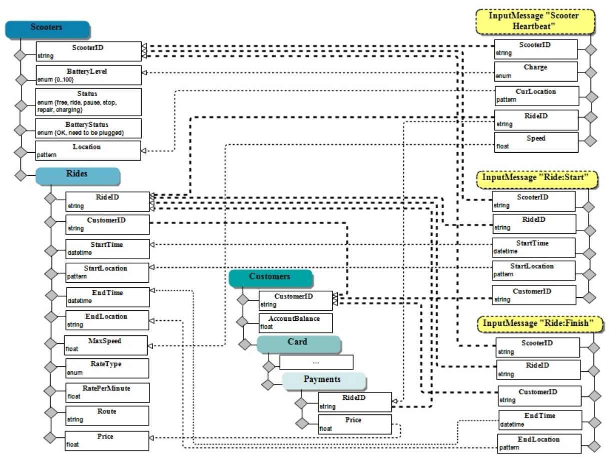

Fig. 2. Data objects.

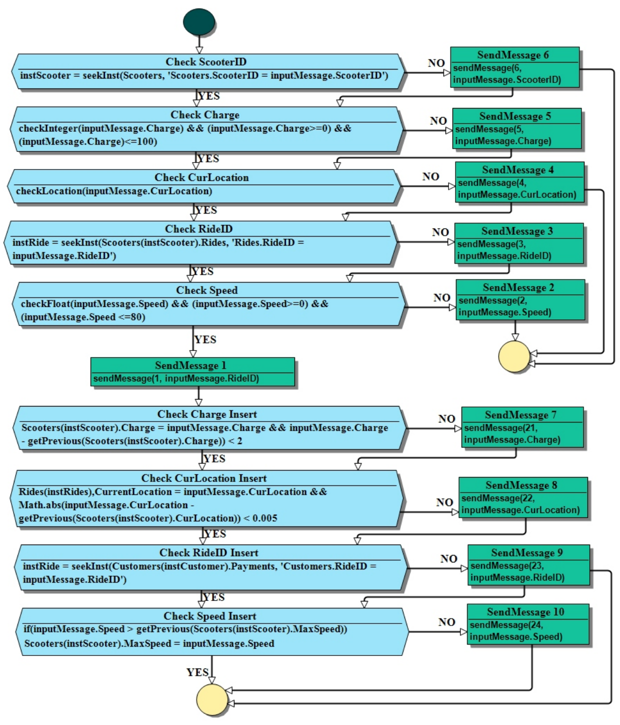
Check ScooterID
instScooter = seekInst(Scooters, 'Scooters.ScooterID = inputMessage.ScooterID')
Check Charge
checkInteger(inputMessage.Charge) && (inputMessage.Charge>=0) && (inputMessage.Charge)<=100)
Check CurLocation
checkLocation(inputMessage.CurLocation)
Check RideID
instRide = seekInst(Scooters(instScooter).Rides, 'Rides.RideID = inputMessage.RideID')
Check Speed
checkFloat(inputMessage.Speed) && (inputMessage.Speed>=0) && (inputMessage.Speed <=80)
SendMessage 1
sendMessage(1, inputMessage.RideID)
SendMessage 2
sendMessage(2, inputMessage.Speed)
SendMessage 3
sendMessage(3, inputMessage.RideID)
SendMessage 4
sendMessage(4, inputMessage.CurLocation)
SendMessage 5
sendMessage(5, inputMessage.Charge)
SendMessage 6
sendMessage(6, inputMessage.ScooterID)
Check Charge Insert
Scooters(instScooter).Charge = inputMessage.Charge && inputMessage.Charge - getPrevious(Scooters(instScooter).Charge)) < 2
Check CurLocation Insert
Rides(instRides).CurrentLocation = inputMessage.CurLocation && Math.abs(inputMessage.CurLocation - getPrevious(Scooters(instScooter).CurLocation)) < 0.005
Check RideID Insert
instRide = seekInst(Customers(instCustomer).Payments, 'Customers.RideID = inputMessage.RideID')
Check Speed Insert
if(inputMessage.Speed > getPrevious(Scooters(instScooter).MaxSpeed)) Scooters(instScooter).MaxSpeed = inputMessage.Speed
SendMessage 7
sendMessage(21, inputMessage.Charge)
SendMessage 8
sendMessage(22, inputMessage.CurLocation)
SendMessage 9
sendMessage(23, inputMessage.RideID)
SendMessage 10
sendMessage(24, inputMessage.Speed)
YES
NO

Fig. 3. Data quality specification.

However, as the system is complex enough and many different cases need to be addressed, in this paper we show only one simplified example when the data on the particular ride is collected, more precisely - *InputMessage* "*Scooter Heartbeat*" in the way of input messages.

The example above analyses the correctness of the data in the *InputMessage* "*Scooter Heartbeat*" data object, however, database data objects can be analysed as well. These checks are simple enough, but it is clear that the verification of the data to be entered is insufficient. Thus, conditions for the correct distribution of data in the database are defined to perform semantic/ contextual checks (Fig. 3), such as:

- whether a scooter to whom *inputMessage* applies exists in the database (by means of *ScooterID*), that is checked by verifying whether ***Scooter.ScooterID = inputMessage.ScooterID****;*
- whether a new instance has been added to the *Scooter* data object *BatteryLevel* data item with a corresponding *Charge* value, which is also between 0 and 100% (positive or "0") and will decrease (higher 2%) or equal to the value previously read since scooter is discharging during the ride (3 checks are hidden here – see the 2nd and 6th boxes of Fig. 3). This is found by looking at the previous record, by addressing it through *ScooterID* un *RideID* that is found by using ***getPrevious()*** function (SQL uses ***LAG*** function for this purpose). This value is important for the *BatteryStatus* attribute since if it is between 0 and 45, the status will be "*need to be plugged*", if greater (46..100) – *"OK"*;

- whether there is a ride to whom *inputMessage* applies - *RideID* of the *Scooter* sub-object *Rides,* that is checked by verifying whether ***Rides.RideID* = *inputMessage.RideID***;
- whether a new *Speed* value is between 0 and 80 and is greater than the value of the *Scooter* data object *MaxSpeed* value – if yes, the *MaxSpeed* value of *Scooter* data object (*Rides* sub-object) should be replaced by *Speed* value of *inputMessage*, otherwise, it is ignored (5th and 9th boxes in Fig. 3).

When both data objects (Fig. 2) and data requirements/ conditions (Fig. 3) are prepared (the step 0 is completed), the DQ-model used to generate tests is derived from them. The nature of all models and checks is in line with the needs we identify using the SUT. This also means that, depending on the system, other more or less specific requirements may be required.

B. Step I: Expanding DQ-model into a Tree

The next step is to expand the data quality conditions in the flowcharts of a tree-like format (Fig. 4). Branch vertexes contain numbers numbering paths, which in the scope of the demonstrated example are 11.

6 of them test the syntactical and contextual correctness of input data (#2-7 nodes), 4 of them test data allocation in the database (#8-11) and another 1 branch #1 represents the correct data processing from syntactical checks of input data to data correct retention in the database.

Every path is in START..END form, the contents of which for 5 branches are presented in Table 1 and are obtained from Fig. 4. Since the test criterion is based on the requirement that all data quality transitions must be walked through, the total number of tests is equal to the number of vertexes – 11. Tests should be developed based on the test criterion, walking along all identified branches. This is done by applying symbolic execution to commands in conditional flowcharts.

C. Step II: Reachability of Branches

The conditions for reaching the branches are derived from the expansion of data quality conditions in the tree (Fig. 4 and Table 1). The conditions for 5 paths of Table 1 are provided in Table 2 demonstrating different cases, where (a) data input and retention in the DB were correct (path #1), (b) input data was invalid (path #2, #5, #6) and (c) allocation of data items in the database was unsuccessful because the corresponding *RideID* was not found in the *Customers* data object.

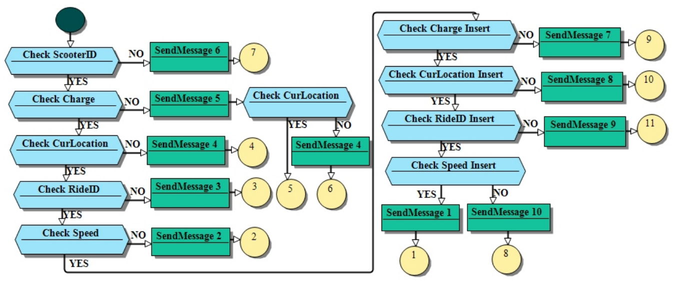

Fig. 4. DQ-model expansion into a tree.

TABLE I. DATA QUALITY EXPANSION IN A TREE

Path	Branch
1	START => Check ScooterID => YES => Check Charge => YES => Check CurLocation => YES => Check RideID => YES => Check Speed => YES => Check Charge Insert => YES => Check CurLocation Insert => YES => Check RideID Insert => YES => Check Speed Insert => YES => SEND Message(1) => END
2	START => Check ScooterID => YES => Check Charge => YES => Check CurLocation => YES => Check RideID => YES => Check Speed => NO => SEND Message(2) => END
…	…
5	START => Check ScooterID => YES => Check Charge => NO => SendMessage(5) => Check CurLocation => YES => END
6	STRAT => Check ScooterID => YES => Check Charge => SendMessage(5) => NO => Check CurLocation => NO => SendMessage(4) => END
…	…
11	START => Check ScooterID => YES => Check Charge => YES => Check CurLocation => YES => Check RideID => YES => Check Speed => YES => Check Charge Insert => Check CurLocation Insert => YES => Check RideID Insert => NO => SEND Message(9) => END

TABLE II. CONDITIONS FOR BRANCHES

Path	Condition
1	1) exist Scooters(instScooter) where (Scooters.ScooterID = inputMessage.ScooterID) 2) valid Value inputMessage.Charge 3) valid Value inputMessage.CurLocation 4)exist Scooters(instScooter).Rides(instRide) where Rides(instRide).RideID= inputMessage.RideID 5) valid Value inputMessage.Speed 6) Scooters(instScooter).BatteryLevel= inputMessage.Charge 7) Scooters(instScooter). Location= inputMessage.curLocation 8) exist Customers(instCustomer).Payments(instRide) where Payments(instPayment).RideID= inputMessage.RideID 9) if(Scooters(instScooter).Rides(instRide) where Rides(instRide).MaxSpeed < inputMessage.Speed) Rides(instRide).MaxSpeed= inputMessage.Speed
2	1) exist Scooters(instScooter) where (Scooters.ScooterID = inputMessage.ScooterID) 2) valid Value inputMessage.Charge 3) valid Value inputMessage.CurLocation 4)exist Scooters(instScooter).Rides(instRide) where Rides(instRide).RideID= inputMessage.RideID 5) **invalid** Value inputMessage.Charge
5	1) exist Scooters(instScooter) where (Scooters.ScooterID = inputMessage.ScooterID) 2) **invalid** Value inputMessage.Charge 3) valid Value inputMessage.CurLocation
6	1) exist Scooters(instScooter) where (Scooters.ScooterID = inputMessage.ScooterID) 2) **invalid** Value inputMessage.Charge 3) **invalid** Value inputMessage.CurLocation
11	1) exist Scooters(instScooter) where (Scooters.ScooterID = inputMessage.ScooterID) 2) valid Value inputMessage.Charge 3) valid Value inputMessage.CurLocation 4) exist Scooters(instScooter).Rides(instRide) where Rides(instRide).RideID= inputMessage.RideID 5) valid Value inputMessage.Speed 6) Scooters(instScooter).BatteryLevel= inputMessage.Charge 7) Scooters(instScooter). Location= inputMessage.curLocation 8) exist Customers(instCustomer).Payments(instRide) where Payments(instPayment).RideID >< inputMessage.RideID

TABLE III. DATA OBJECT INPUTMESSAGE "SCOOTER HEARTBEAT"

Test #	Scooter ID	Charge	Cur Loc	Ride ID	Speed	Msg. #
1	scooter-1	charge-1	loc-1	ride-1	speed-1	1
2	scooter-2	charge-2	loc-2	ride-2	speed-2 **invalid**	2
3	scooter-3	charge-3	loc-3	ride-3 **invalid**	-	3
4	scooter-4	charge-4	loc-4 **invalid**	-	-	4
5	scooter-5	charge-5 **invalid**	loc-5	-	-	5
6	scooter-6	charge-6 **invalid**	loc-6 **invalid**	-	-	5, 4
7	scooter-7 **invalid**	-	-	-	-	6
8	scooter-8	charge-8	loc-8	ride-8	speed-8	10
9	scooter-9	charge-9	loc-9	ride-9	speed-9	7
10	scooter-10	charge-10	loc-10	ride-10	speed-10	8
11	scooter-11	charge-11	loc-11	ride-11	speed-11	9

This is used in the scope of the 2nd step of the presented algorithm – for each tree branch, the condition for path feasibility shall be established by means of symbolic execution (partly covered in the next subsection).

By resolving the conditions for path feasibility in all 11 cases, test input data and data objects content are obtained on which full coverage of the conditions of the data quality model can be obtained by the execution of these conditions.

D. *Step III: Preparing a DQ-complete Test Set*

The test generator prepares tests, unless otherwise specified in the conditions, with unique values. They are provided in Table 3 with symbolic names ('-' refers to the lack of the need to define the symbolic value as this attribute is not covered in the particular test). The content of data objects for all 11 branches is provided. Table 3 representing *InputMessage* is divided into 2 parts – (1) syntactic checks (tests #1..7) and (2) data allocation in the DB (tests #8..11).

Verification of context conditions is ensured by including instances with relevant key values in data object. For instance, in order to ensure the *Check ScooterID* check in all 11 tests, the *Scooter* data object includes 10 instances with different key values (*scooter-1, scooter-2, ...scooter-6, scooter-8,..., scooter-11*), but for the 7th test, *Scooter* data object does not contain an instance with an appropriate key value. Similarly, the *Rides* data object contains 6 instances with key values corresponding to tests, i.e. *ride-1, ride-2, ride-8, ride-9, ride-10 and ride-11*. Attribute values for syntactic checks (tests #1…7) are generated according to the specification for data checks, allowing task-specific checks, which in the case of provided example are *Check Location*, *Check Speed* etc.

The generated test set and data object values are the DQ-complete test set. Because the DQ-model is executable, its execution with a DQ-complete test set (input data and database generated content) will result in a protocol that appears in Table 4.

TABLE IV. TESTING RESULTS

Path #	Message #	Message text
1	*1*	**successful input**: <*scooter-1, charge-1, loc-1, ride-1, speed-1*>
2	*2*	**input error: invalid speed** < *scooter-1, charge-1, loc-1, ride-1, speed-1*>
3	*3*	**input error: invalid RideID** <*scooter-3, charge-3, loc-3, ride-3*>
4	*4*	**input error: invalid CurLocation** <*scooter-4, charge-4, loc-4*>
5	*5*	**input error: invalid Charge** <*scooter-5, charge-5*>
6	*4*	**input error: invalid CurLocation, Charge** < *scooter-6, charge-6, loc-6*>
7	*6*	**input error: invalid ScooterID** <*scooter-7*>
8	*10*	**database error: Speed is not inserted** <*scooter-8, charge-8, loc-8, ride-8, speed-8*>
9	*7*	**database error: Charge is not inserted** <*scooter-9, charge-9*>
10	*8*	**database error: curLocation is not inserted** <*scooter-10, charge-10, loc-10*>
11	*9*	**database error: RideID is not inserted** <*scooter-11, charge-11, loc-11, ride-11*>

It covers all cases in which testing we were initially interested in, detecting defects including the above mentioned, more precisely, inaccurate data on charge and current location.

If the SUT is tested with a prepared generated complete set of tests, the result must, in substance, correspond to the result of the execution of the DQ-model. Thus, it can be argued that the test objective has been achieved – the tested item has been tested with input data ensuring that all data quality conditions, and the DB content are checked. This also means that the tester does not have to prepare the test by himself and perform the execution of the SUT with them, thus, the quality of testing does not depend on the qualification of the tester, but on the quality of the DQ-model, i.e. how accurate the testing model meets the requirements of the system.

V. CONCLUSIONS

The paper proposes the Data Quality Model Based Testing (DQMBT) approach of IS, that uses previously proposed data quality model as a testing model by demonstrating it through specific example – the system of e-scooters that is currently in use. The proposed solution is based on the new specific criterion of a complete testing verifying the correctness of all input data and their retention in the DB with tests containing all possible conditions for input data values. This is achieved through symbolic execution, which significantly reduces the number of tests, thereby allowing to perform a finite number of tests covering [almost] infinite set of possible cases.

The proposed algorithm first checks the compatibility of the data to be entered into the system with the syntactical and contextual conditions, defined in DQ-model. Correct data are stored in data objects. They shall then be checked for compliance with the input data, by which the correctness of data retention is tested. Thus, the data are examined (1) first as the compliance of the input data with the syntax and context conditions, (2) whether the data are correctly stored in the database and the stored data corresponds to those entered. The proposed DQMBT approach uses a formal executable specification, generating the DQ-complete test set and the expected results of its execution. When automated testing of the system is completed, the tester should only compare the results obtained with the benchmarks. It is obvious that the proposed approach differs vitally from the 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 system. It also significantly reduces tester involvement and workload.

While the solution is a new technology for testing and software development, the provided example covers the case when already developed system is under test. However, it can be used while developing a new system as well.

This time a limited example of one input message was demonstrated, however, it is clear that, in the real circumstances, a number of input messages from different sources (e-scooter, smartphone etc.) may come in the system [almost] simultaneously and their sequence/ order affects the testing process and results. Therefore, further studies will address this question by demonstrating how the proposed approach applies to a number of input messages. The quantitative results of the application of the proposed approach to the IS of e-scooters, compared to the testing technique currently in use, will also be provided in the further studies.

ACKNOWLEDGMENT

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".

REFERENCES

[1] S. Tahvili, M. Saadatmand, M. Bohlin, W. Afzal, S. H. Ameerjan, “Towards execution time prediction for manual test cases from test specification”, In *43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2017, pp. 421-425. IEEE.

[2] V. Garousi, M. V. Mäntylä, “When and what to automate in software testing? A multi-vocal literature review”, *Information and Software Technology*, *76*, 2016, pp. 92-117.

[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,* 2020, pp. 322-329, DOI: 10.5220/0009459703220329.

[4] 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)*, 2019, pp. 303-308. IEEE, DOI: 10.1109/SNAMS.2019.8931867.

[5] S. Kim, R. P. Del Castillo, I. Caballero, J. Lee, C. Lee, D. Lee ... & A. Mate, “Extending Data Quality Management for Smart Connected Product Operations”. *IEEE Access*, 2019, pp. 144663-144678, DOI:10.1109/ACCESS.2019.2945124.

[6] R. Perez-Castillo, A. G. Carretero, I. Caballero, M. Rodriguez, M. Piattini, A. Mate, ... & D. Lee, “DAQUA-MASS: An ISO 8000-61 based data quality management methodology for sensor data. Sensors, 2018, 18(9), 3105, https://doi.org/10.3390/s18093105.

[7] R. Casado-Vara, F. de la Prieta, J. Prieto, J. M. Corchado, “Blockchain framework for IoT data quality via edge computing”, In *Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems*, 2018, pp. 19-24, https://doi.org/10.1145/3282278.3282282.

[8] A. Karkouch, H. Mousannif, H. Al Moatassime, T. Noel, “Data quality in internet of things: A state-of-the-art survey”, *Journal of Network and Computer Applications*, *73*, 2016, pp. 57-81.

[9] A. Nikiforova, “Definition and Evaluation of Data Quality: User-Oriented Data Object-Driven Approach to Data Quality Assessment”, *Baltic Journal of Modern Computing*, *8*(3), 2020, pp. 391-432.

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

[11] J. Bicevskis, Z. Bicevska, A. Nikiforova, I. Oditis, “An Approach to Data Quality Evaluation”, *In 2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 2018, pp. 196-201, IEEE, DOI: 10.1109/SNAMS.2018.8554915.

[12] J. C. King, “Symbolic execution and program testing”, *Communications of the ACM*, *19*(7), 1976, pp. 385-394.

[13] M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, A. Pretschner, “Model-based testing of reactive systems”, In *Vol. 3472 of Springer LNCS,* 2005.

[14] P. Godefroid, “Test generation using symbolic execution”, In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[15] B. Nielsen, A. Skou, “Automated test generation from timed automata”, In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2001, pp. 343-357. Springer, Berlin, Heidelberg.

[16] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, K. Scholl, “Model-based test case generation for smart cards”, *Electronic Notes in Theoretical Computer Science*, *80*, 2003, pp. 170-184.

[17] B. Legeard, F. Peureux, M. Utting, “Automated boundary testing from Z and B”. In *International Symposium of Formal Methods Europe*, 2002, pp. 21-40, Springer, Berlin, Heidelberg.

[18] V. Rusu, L. Du Bousquet, T. Jéron, “An approach to symbolic test generation”, In *International Conference on Integrated Formal Methods*, 2000, pp. 338-357, Springer, Berlin, Heidelberg.

[19] L. Frantzen, J. Tretmans, T. A. Willemse, “A symbolic framework for model-based testing”, In *Formal approaches to software testing and runtime verification*, 2006, pp. 40-54, Springer, Berlin, Heidelberg.

[20] B. Selic, “The theory and practice of modeling language design for model-based software engineering—a personal perspective”. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, 2009, pp. 290-321, Springer, Berlin, Heidelberg.

[21] J. Bicevskis, Z. Bicevska, A. Nikiforova, I. Oditis, I. (2020). *Data Quality Model-based Testing of Information Systems*. In 2020 15th Conference on Computer Science and Information Systems (FedCSIS) (pp. 595-602). IEEE, doi: 10.15439/2020F25

[22] J. Peleska, W. L. Huang, F. Hübner, “Complete Model-based Testing”, *Test, Analyse und Verifikation von Software-gestern, heute, morgen*, 2017, pp. 81-92.

[23] A. Spillner, M. Winter, A. Pietschker, “Test, Analyse und Verifikation von Software–gestern, heute, morgen”, *BoD–Books on Demand*, 2018.

[24] F. Lia, R. Nocerino, C. Bresciani, A. Colorni Vitale, A. Luè, “Promotion of E-bikes for delivery of goods in European urban areas: an Italian case study”, In *Transport Research Arena (TRA) 5th Conference: Transport Solutions from Research to Deployment*, 2014, pp. 1-10.

[25] W. Espinoza, M. Howard, J. Lane, P. Van Hentenryck, “Shared E-scooters: Business, Pleasure, or Transit?”, 2019, arXiv:1910.05807.

[26] A. Brown, N. J. Klein, C. Thigpen, N. Williams, “Impeding access: The frequency and characteristics of improper scooter, bike, and car parking”, *Transportation Research Interdisciplinary Perspectives*, 2020, https://doi.org/10.1016/j.trip.2020.100099