Data-Driven Testing using TTCN-3

Bernard Stepien, Liam Peyton
School of Engineering and Computer Science
University of Ottawa
Ottawa, Canada
Email: {bstepien | lpeyton}@uottawa.ca

Mohamed Alhaj
Computer Engineering Department
Al-Ahliyya Amman University
Amman, Jordan
Email: m.alhaj@ammanu.edu.jo

***Abstract*—Complex software systems orchestrate interactions between components of the system. Integration testing of such systems involves making individual unit tests for individual components that work together to test the interactions between components. Unit tests for different components often consist of heterogeneous representations of test data and test behavior written in various implementation languages. As a result, in integration testing it is an advantage to use a single formal testing language like TTCN-3 (Testing and Test Control Notation Version 3). We propose a transformation tool for Data-Driven Testing to generate TTCN-3 test suites that include data types, templates and test behavior from tables. This process is relatively straightforward for relational data bases and XML (eXtensible Markup Language) because they are based on well-defined data models. Excel is more complex because it has no such data models. We have developed a tool that assists the tester in extracting TTCN-3 typing information from Excel tables to produce TTCN-3 templates and test behaviors and optimize their re-usability.**

Keywords: Data-driven Testing; Testing; TTCN-3; re-usability.

I. INTRODUCTION

Complex software systems orchestrate interactions between components of the system. Integration testing involves making individual unit tests for individual components that work together to test the interactions between components. Unit testing alone does not guarantee that components interact correctly. Unit tests for different components often consist of heterogeneous representations of test data and test behavior written in various implementation languages. Ideally, integration testing would use a single formal testing language like TTCN-3 (Testing and Test Control Notation Version 3).

Data-Driven Testing (DDT) is well known in industry. There are a variety of industry-oriented definitions online and the concept is discussed and explained in detail in Web sites [4][6], user forums [5][11], frameworks [9][12], patents [13][14], application domains [10] and linked with other testing models [3][15]. The basic principle consists of separating test data (inputs and expected outputs) from test scripts (test behavior) as shown in Figure 1. The test data is stored as tables in relational databases, XML (eXtensible Markup Language) documents or Excel spreadsheets. More advanced test technologies such as TTCN-3 [3] allow a flexible separation of concerns between an abstract layer that consists of test data and test logic and a concrete layer that consists of codecs to encode and decode data into the specific format and protocol needed to test a component. In particular, the TTCN-3 concept of template to represent test data and expected responses is reusable whereas simple DDT is not, and TTCN-3 strong typing enables early detection of errors in test data.

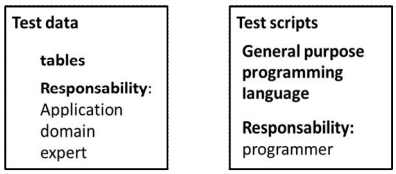

Figure 1. DDT separation model

Thus, we propose a transformation tool to generate TTCN-3 test suites that include data types, templates and test behavior, from DDT tables. This process is relatively straightforward for relational data bases and XML because they are based on well-defined data models. However, the case of Excel [1] is more complex because such data models do not exist. We have developed a tool that assists the tester in extracting TTCN-3 typing information from Excel tables to produce TTCN-3 templates and test behavior and optimize their re-usability.

The rest of the paper is structured as follows. In section II we present an overview of data-driven testing and TTCN-3. In section III, we present our approach for transforming data-driven test tables into TTCN-3 test suites. In section IV, we present our tool implementation and evaluation. And finally, in section V, we present the conclusion.

II. DATA-DRIVEN TESTING AND TTCN-3

The main goal of DDT is to allow application domain experts without programming skills to prepare test data and to reduce maintenance costs. Test data is commonly stored in tables using one of the following three mechanisms:

- Relational databases
- XML documents
- Excel tables

While the two first approaches provide data models (table column descriptions for relational databases and XML schema for XML documents) and are thus unambiguous, Excel spreadsheets do not. The data models are absent

 ISBN: 978-1-61208-632-3

because tables contain only data with column headings. Although one can set data types for the cells of a column mostly to specify the display format for numeric types (number of decimal digits for numbers), the default data type is the *general* data type. Also, there is no explicit definition of field names. Only column headings hint at what the fields in a structured data type could be.

Another challenge in DDT is that tests are strictly sequential as it is impossible to describe alternatives easily with tables only. Thus, a test step consists of reading a row of data, performing the test by either sending a message to the system under test (SUT) or invoking a function with parameters and checking the response message of the SUT or the return value of the function against a test oracle (expected response). It is the responsibility of the programmer of the test script to determine the exact location of the various pieces of data in the table to transfer them to the fields of some structured type variable and distinguish what is test data from what is a test oracle.

The test scripting language TTCN-3 has been used for model-driven testing in general, and has many features that make it an effective tool for DDT. TTCN-3 is based on a separation of concerns between an abstract layer and a concrete layer. The basic elements of an abstract layer consist mainly in the following components:

- Data typing definitions
- Templates definitions
- Behavior definitions

As shown in Figure 2, separate template definitions for Test Data, and separate test behavior definitions for test scripts means that TTCN-3 has the same separation model that DDT has (as shown in Figure1).

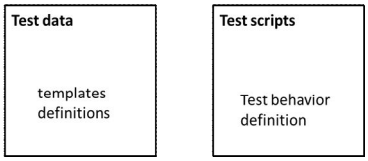

Figure 2. TTCN-3 separation model

A TTCN-3 template defines test data (stimuli or test oracle). Each template has a name that can be referred to in behavior definitions or reused in further template definitions like a variable. For test oracles, that variable contains program code used to verify that a response corresponds to the test oracle. The concrete layer consists of codecs that translate abstract into concrete data and vice versa and communicates with the SUT. We present three examples next.

Data type definition example for a structured type

The main difference with Excel-based DDT is that in TTCN-3 we define data types as shown in Figure 3 to be able to define templates. This is because in TTCN-3, the matching of test oracles is achieved at once for all test data as opposed to the DDT approach of using an atomic assertion mechanism for each individual piece of data .

```
type record MyCarRequestType {
   integer nbDoors,
   charstring model,
   charstring brand }

type record MyCarResponseType {
   charstring model,
   charstring brand,
   float listPrice }
```

Figure 3. TTCN-3 Data Typing Example

XML or Database [16] based DDT is handled via a built-in mechanism of TTCN-3 tools that translates for example an XML schema directly into TTCN-3 data types.

Template definition example

A TTCN-3 template as show in Figure 4 resembles a structured type variable assignment but in essence it is very different from a typical programming language variable. The values being assigned to the fields of this structured data type have two different meanings depending of the direction of a message in the communication system. When using the template for sending data, they are plain data that is either encoded to be sent in the case of messages or values of parameters for a function being invoked. When using the template as a test oracle, the values mean that the response message or return value must match the values given in the template. The matching mechanism itself is a built-in feature of TTCN-3 execution tools and thus does not need to be programmed by the users. Thus, a TTCN-3 template is more like an implicit program.

```
template MyCarRequestType
              myTiguanRequest := {
   nbDoors := 5,
   model := "Tiguan",
   brand := "VW"
}

template MyCarResponseType
               myTiguanResponse := {
   model := "Tiguan",
   brand := "VW",
   listPrice := 35000.00
}
```

Figure 4. TTCN-3 Template Example

Behavior definition example

Behavior definitions as shown on Figure 5 consist in sending data to the SUT and trying to match a response or return value to a test oracle. The TTCN-3 *send* and *receive* commands use template names where data or test oracles are defined. TTCN-3 *receive* statements are usually contained in an *alt* statement (alternative). This is to handle various potential responses and assign a corresponding verdict (pass or fail). The generic *receive* without parameters means

receive any value and tester typically assign a fail verdict with such a construct.

```
myPort.send(myTiguanRequest);
timer myTimer = 5.0;
alt {
   [] myPort.receive(myTiguanResponse)
         { setverdict (pass)}
    [] myPort.receive
         {setverdict(fail)}
    [] myTimer.timeout
         {setverdict(inconc)}
}
```

Figure 5. TTCN-3 Test Behavior Example

TTCN-3 also has timers that can be set and timeouts are part of an alternative. If any of the receive statements in the alternative do not match the response, eventually the timer will time out and a corresponding verdict can be set. Also, the receive statement is not fully equivalent to an assertion. When a receive statement fails, TTCN-3 merely tries the next alternative. This is similar to a rule based system.

Because a template is like a variable, it is fully re-usable either in different tests but also in the definition of other templates where a field is of the data type of the re-usable template. Another interesting aspect of templates is that since templates are referenced by name, when performing tests with the same data, it doesn't need to be redefined or read for each test like in DDT. More important is the feature that allows deriving a template from another existing template by specifying only the delta, thus avoiding specifying portions of the same data several times.

```
template MyType myGolfRequest
     modifies myTiguanRequest := {
              model:= "Golf" }
```

Figure 6. TTCN-3 template modification Example

Transforming DDT into TTCN-3 has several benefits. From a language point of view, TTCN-3 is based on strong typing. Strong typing allows one to restrict the usage of data by type. In other constructs, such as templates data, being sent or received can be set to a precise type. In loose table formats such as Excel, there is no way to specify such restrictions which inevitably leads to undetectable errors at design time. Relational databases or XML documents are typed but not always strongly. For example in relation databases there is no way to specify exactly which values are allowed in a specific data type. In our *MyCarRequestType*, we could have further refined this type definition by restricting the brand field type. Instead of using the generic *charstring* type, we could have defined a *brandType* as follows:

```
type charstring brandType
( "VW", "Mercedes", "Renault", "Fiat",
"Ford", "Chrysler" );
```

Figure 7. TTCN-3 type Restriction Example

Then this *brandType* could have been used in the *MyCarRequestType* as follows:

```
type record MyCarRequestType { integer
nbDoors, charstring model, brandType
brand }
```

Figure 8. TTCN-3 data sub-typing Example

The use of a brand name, other than the one found in the list of the data type *brandType,* would cause a compile error. In DDT, the same error would be detected only at run time. The following example would trigger a compile error.

```
template MyCarRequestType
myToyotaRequest := {
   nbDoors := 5,
   model := "Corolla",
   brand := "Toyota"
}
```

Figure 9. TTCN-3 template with Restricted sub-type Example

The other benefit of TTCN-3 is in its test results display. Each test event (send or receive) is displayed and TTCN-3 tools allow for inspection of the results by providing a comparison between the response data received and the test oracle as shown in Figure 10 where the expected *listPrice* of $35000.00 did not match the response value of $15000.00.

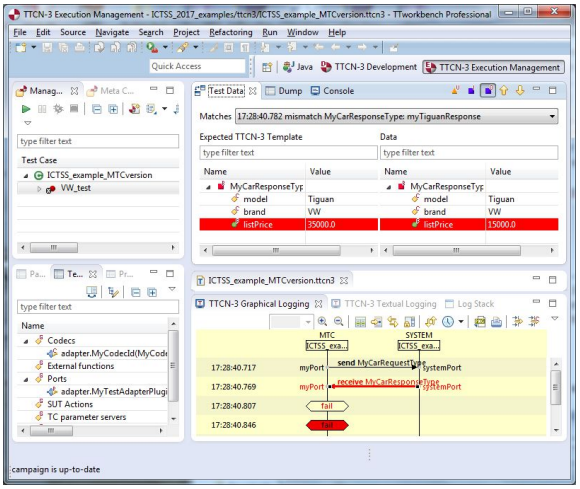

Figure 10. TTCN-3 tools results inspection feature

Transforming relational data bases into TTCN-3 have already been handled by Stepien et al. [7, 8]. They are also supported by most TTCN-3 tools. However, until now, the conversion of Excel tables into TTCN-3 has not been addressed in TTCN-3 or the academic literature.

III. TRANSFORMING TABLES INTO TTCN-3 TEST SUITES

Transforming Excel tables into TTCN-3 test suites consists of determining data types which include field names of the implicit structured type that a table represents and the type of each such field. Also, we need to distinguish what is data to be sent from data that represents a test oracle.

 ISBN: 978-1-61208-632-3

For example, in the Excel table shown in Figure 11, we can find two sub-tables, one for stimuli test data and one for response test oracles. The stimuli sub-table is a simple structured type while the response test oracle table is a complex structured type where the observations field is itself a structured type.

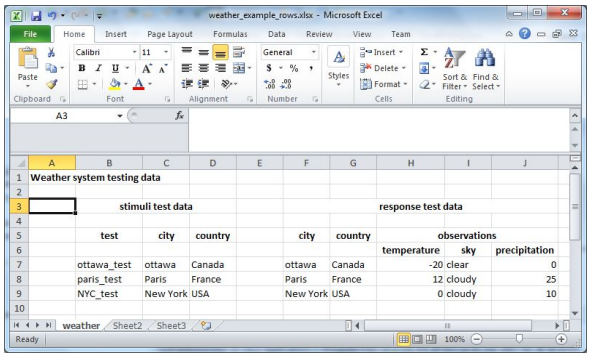

Figure 11. Excel table to be converted example

The problem is how to automate the process of determining the data types and location where to read data and then transforming them into TTCN-3. While there have been cases of Excel tables converted to TTCN-3 in some industrial projects, there are no publications about the process because typically each Excel spreadsheet was handled manually on an ad hoc basis to determine data type and where to read the appropriate data.

We have approached this conversion problem in two different ways: first we considered a fully automated conversion using principles of artificial intelligence where the system would locate the table of data automatically by for example discovering that a column contains data of the same data type, then consider the data found in the rows preceding the data as headings and any other loose and isolated row as comments. However, one major problem with this approach is that there is no indication in Excel tables as to what a stimulus is and what a response is. This results in inconsistencies. In a second approach we have used an interactive mock-up of the Excel table for the tester to delimit the portions of the table that corresponds to either column heading, stimuli data and response data. This gives the tester control over the specification of stimuli and responses.

Such a tool is more efficient than the traditional approach of hard coding the locations of data in test logic and creating the data type definitions manually. Also, this process is of value when considering economies of scale with large numbers of Excel tables. It large projects with extensive use of DDT there could be tens of thousands of such tables.

This is a fundamental choice based on the principle of strong typing. Effectively, if we would follow the DDT model of reading data from tables and applying them to the test script directly, we would detect errors in tables at run time only. This inevitably increases the testing cycle where tests have to be run several times and test results analyzed. By comparison, the TTCN-3 template approach would detect a number of errors already at compile time when the converted templates are compiled.

Also, if the process is fully automated, the user, in this case the application domain expert, not the programmer, can correct the errors in the Excel table and the TTCN-3 test suite can be automatically re-generated and thus re-compiled without any additional efforts from the programmer. It has to be noted that the original DDT Excel table approach is not completely eliminated because it is still a benefit to have a non-programmer domain expert to code test data. Actually, with this automated Excel table conversion process, the coding effort of the programmer is quasi null. The only task for the TTCN-3 programmer is to direct the application domain expert to the elements in the Excel table which have errors.

A. Extracting TTCN-3 Typing from Excel Spreadsheets

TTCN-3 typing can be derived from the tables quasi automatically. The data can be scanned to determine their type (alphanumeric, numeric or boolean). Also, the field names of a structured data type can be derived from the headings of the columns as for example in the range of row/column B5 to J6 in Figure 11. Complex data types containing fields that are themselves of a structured sub-type can be derived using the indication of Excel spans of a cell, here in cell H5 for the *observation* field that covers the range H6 to J6 for the field names of this sub-type. The generated data types contain comments that indicate their origin on the table to improve traceability.

```
type record StimuliType {
    charstring city,        // cell C5
    charstring country      // cell D5
}

type record ResponseType{
    charstring city,        // cell F5
    charstring country,     // cell G5
    ObservationType observations
                            // cell H5
}

type record ObservationType {
    float temperature,     // cell H6
    charstring sky,        // cell I6
    integer precipitation // cell J6
}
```

Figure 12. TTCN-3 Generated Datatypes Example

B. Generating TTCN-3 templates

Each piece of data of a table is assigned the value of a field of a template. Each row of the table generates separate templates in addition to separate templates for stimuli and response test oracles as follows:

 ISBN: 978-1-61208-632-3

```
template StimuliType ottawa_test_stimuli
:= {
    city := "Ottawa",      // cell C7
    country := "Canada"    // cell D7
}

template ResponseType
           ottawa_test_response := {
    city := "Ottawa",      // cell F7
    country := "Canada",   // cell G7
    observations := {
       temperature := -20,// cell H7
       sky := "cloudy",    // cell I7
       precipitation := 0 // cell J7
}    }
```

Figure 13. TTCN-3 Generated Templates Example

C. *Generating test behavior*

Finally, DDT tables can be interpreted as behavior of the sequential form unless indicated as shown in Figure 14.

```
testcase weather_service_test()
runs on MTCType  system SystemType {
   timer myTimer :=5.0;
   map(mtc:myPort, system:systemPort)
   // row 7
   myPort.send(ottawa_test_stimuli);
   alt {
     [] myPort.receive
           (ottawa_test_response){
         setverdict (pass) }
     [] myPort.receive
         {setverdict(fail)}
     [] myTimer.timeout
         {setverdict(inconc)}
   }
   // row 8
   myPort.send(paris_test_stimuli);
   alt {
      [] myPort.receive
              (paris_test_response){
                setverdict (pass)}
      [] myPort.receive
               {setverdict(fail)}
      [] myTimer.timeout
               {setverdict(inconc)}
   }
   // row 9
   myPort.send(NYC_test_stimuli);
   alt {
      [] myPort.receive
                 (NYC_test_response){
             setverdict (pass) }
      [] myPort.receive
            {setverdict(fail)}
      [] myTimer.timeout
            {setverdict(inconc)}
   }
   unmap(mtc:myPort, system:systemPort);
}
```

Figure 14. TTCN-3 Generated Test Behavior Example

Thus, each row can produce a stimuli being sent and an alternative of a response test oracle with both any value and timeout alternatives. Here again for traceability reasons, we show the row number in the table that corresponds to the test step. If we generate templates with names found in the column with the heading *test* like *ottawa_test*, the table shown in Figure 11 would generate the test behavior shown in Figure 14. The advantage of a TTCN-3 template approach for conducting DDT is that everything is clearly defined and thus is easily traceable at run time without having to go through trace stacks.

IV. TOOL IMPLEMENTATION AND EVALUATION

We have developed and validated these techniques in the testing of an avionics software system. In particular, we implemented a tool to automate the transformation of the tables into a TTCN-3 test suite. As can be seen in Table [15] and, the tool provides an interactive marking mechanism. Each portion of the table can be highlighted and a pull down menu provides categories to choose from in order to indicate how to use the selected portion of cells to the tool. There are three categories of markings required to generate a correct TTCN-3 test suite:

- Delimiting column headings to be used as field names for structured data types code generation
- Delimiting the two sub-tables of stimuli and response test oracles for templates code generation
- Delimiting the test names column if present to generate template names.

Our marking tool is a mock-up of the Excel spread-sheet in that it shows the rows and columns with the content of the cells as placed in the spread sheet. However, these cells are used for only one purpose, delimiting each zone according to their functionality in the TTCN-3 code generation. No other functionality, like calculations provided by the Excel sheets, can be performed. Also, the code generation makes use mostly of combinations of such markings.

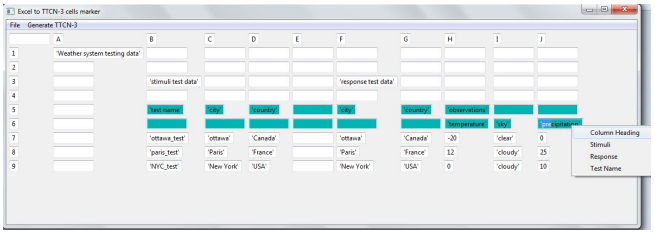

Figure 15. Delimiting column headings

For example, the marking of column headings shown in Figure 15 is not enough for generating data types because there are two separate groups of data types to be defined, one for stimuli and one for response test oracles. Thus, one must separate the table, shown in Figure 16, and select the portions of the table that belong to either stimuli or

responses. This includes the column headings since both data types and test data need to be separated into stimuli and responses.

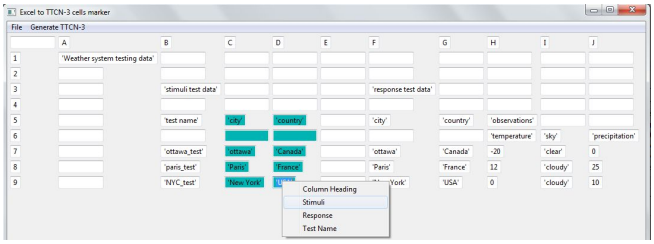

Figure 16. Delimiting the stimuli sub-table

Manual creation of test scripts in TTCN-3 to execute the tables before the tool was implemented took on average one day per test script. With the tool a complete suite of test scripts was created in one hour. As well, the manual process was error-prone and inconsistent whereas the automated scripts were standardized and needed far less maintenance.

From an implementation point of view, it might have been ideal to use Excel for the highlighting and subsequent export to TTCN-3. However, the export would depend on what commercial tool is available. Thus, we decided on a model to convert the table into a two dimensional array or more precisely different parallel arrays, one containing the data itself and others containing properties such as data types or formatting instructions such as spans that are important to detect sub-structured data types.

Finally, our tool produces only the abstract test suite in TTCN-3. The concrete layer of codecs and communication software specific to the application domain needs to be in place. This is built once (based on TTCN-3 abstract data types) and is reusable by any test suite generated by our tool. This provides a structured approach with a clean separation of concerns (abstract tests vs domain-specific coding/encoding) enabling full re-usability. Traditional unit-testing, by comparison tends to mix test event checking with coding/decoding and communication activity in an ad hoc manner that does not facilitate re-use.

V. CONCLUSION

DDT is an important testing approach for generation and automation of test campaigns. For such benefits to scale it is important that such generation and automation be systematic and strongly-typed. It is also important that the full complexity of parallel test scripts be supported. TTCN-3 provides strong features to support such an approach to TTCN-3 and we have demonstrated how it can be integrated and applied even when the approach to DDT specifications is relatively low-tech and ad hoc through the use of Excel tables. Our approach and tool prototype greatly reduced the manual effort in generating test campaigns, allowed flexible support of Excel for non-technical testers while integrating standardization, strong type and parallel text execution with TTCN-3.

ACKNOWLEDGMENT

The authors would like to thank the Spirent company for providing us the necessary tool *TTworkbench* to carry out this research as well as funding from CRIAQ, MITACS and ISONEO SOLUTIONS for which this research has been conducted.

REFERENCES

[1] Microsoft Excel, 2018. Accessed March 2018 at https://support.office.com/en-us/excel

[2] ETSI ES 201 873-1, The Testing and Test Control Notation version 3 Part 1: TTCN-3 Core Language, May 2017. Accessed March 2018 at http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.09.01_60/es_20187301v040901p.pdf

[3] P. Shinde and A. Sathe, Data-Driven Software Testing for Agile Development, in PhUSE 2011, Brighton, United Kingdom, June, 2011. Accesed March 2018 at https://www.lexjansen.com/phuse/2011/ts/TS08.pdf

[4] Microsoft Corporation, How To: Create a Data-Driven Unit Test in Visual Studio, 2015. Accessed March 2018 at https://msdn.microsoft.com/en-us/library/ms182527.aspx

[5] StackExchange, What are some good approaches to separating test data from test scripts, 2013. Accessed March 2018 at https://sqa.stackexchange.com/questions/6678/what-are-some-good-approaches-to-separating-test-data-from-test-scripts

[6] Smartbear, Introduction to Data-Driven Testing, 2018. Accessed March 2018 at https://smartbear.com/learn/automated-testing/introduction-to-data-driven-testing

[7] I. Schieferdecker and B. Stepien, Automated Testing of XML/SOAP Based Web Services. In: Irmscher K., Fähnrich KP. (eds) Kommunikation in Verteilten Systemen (KiVS). Informatik aktuell. Springer, Berlin, Heidelberg, 2003. https://doi.org/10.1007/978-3-642-55569-5_4

[8] B. Stepien and L. Peyton, Integration Testing of Web Applications and Databases Using TTCN-3. In: Babin G., Kropf P., Weiss M. (eds) E-Technologies: Innovation in an Open World. MCETECH 2009. Lecture Notes in Business Information Processing, vol 26. Springer, Berlin, Heidelberg, 2009. https://doi.org/10.1007/978-3-642-01187-0_26

[9] Chandrapabha, A. Kumar and S. Saxena, Data Driven Testing Framework using Selenium WebDriver, in International Journal of Computer Applications (0975-8887) vol. 118-No. 18, May 2015. Accessed March 2018 at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.695.9076&rep=rep1&type=pdf

[10] E. G. Gomez, M. Casado, M. Stanka and S. Korner, Automated Regression Testing of Complex Mission Control Applications, SpaceOps 2010, Huntsville, Alabama, April 2010. Accessed March 2018 at https://arc.aiaa.org/doi/pdf/10.2514/6.2010-2289

[11] D. Cheney, Writing Table Driven Tests in Go, 2013. Accessed March 2018 at https://dave.cheney.net/2013/06/09/writing-table-driven-tests-in-go

[12] N. Daley, D. Hoffman and P. Strooper, A framework for table driven testing of Java classes. Softw: Pract. Exper., 32: 465–493, 2002. doi:10.1002/spe.452. Accessed March 2018 at http://onlinelibrary.wiley.com/doi/10.1002/spe.452/full

[13] J. S. Schaefer, Systems and methods for table driven automation testing of software programs, Capital One Financial Corporation, 2006. U.S. Patent 6,993,748. Accessed March 2018 at https://patents.google.com/patent/US6993748B2/en

[14] J. J. Haswell, R. J. Young, and K. Schramm, System, method and article of manufacture for a table-driven automated scripting

 ISBN: 978-1-61208-632-3

architecture, Accenture Llp, 2002. U.S. Patent 6,502,102. Accessed March 2018 at https://patents.google.com/patent/US6502102B1/en

[15] J. Zander, Z. R. Dai, I. Schieferdecker and G. Din, From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing -. In: Khendek F., Dssouli R. (eds) Testing of Communicating Systems. TestCom 2005. Lecture Notes in Computer Science, vol 3502. Springer, Berlin, Heidelberg, 2005. https://doi.org/10.1007/11430230_20

[16] G. Adamis, A. Wu-Hen-Chang, G. Németh, L. Eros and G. Kovacs, Data Flow Testing in TTCN-3 with a relational Database Schema, in International SDL Forum, SDL 2013: Model-Driven Dependability Engineering pp 1-18, Springer Verlag

 ISBN: 978-1-61208-632-3