Visualizing Execution Models and Testing Results

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

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

***Abstract*—Software engineering models typically support some form of graphic visualization. Similarly, testing results are shown as execution traces that testing tools, such as TTCN-3 can display as message sequence charts. However, all TTCN-3 tools avoid presenting data directly in the message sequence chart because some of it may be complex structured data. Instead, they simply display the data types used. The real data is made available through detailed message inspection representations when the datatype shown is clicked on. Thus, validation of test results requires a tedious message by message inspection especially for large tests involving sequences of several hundred test events. We propose the capability to specify which data can be displayed in the test results message sequence chart. This provides overview capabilities and improves the navigation of test results. The approach is illustrated with an example of SIP protocol testing and an example of testing an avionics flight management system.**

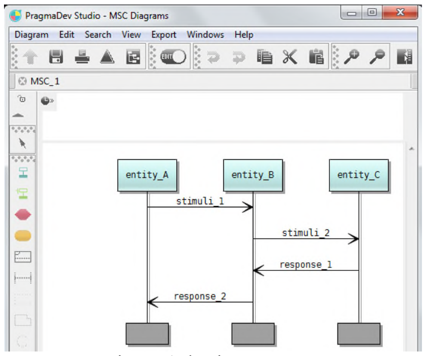
Keywords-sofware modelling; testing; TTCN-3.

I. MOTIVATION

Modeling and testing of software applications are intricately linked. The first describes the expected behavior while the second describes a trace of real behavior of a system. The first preoccupation of a software engineer is to ensure that both expected and actual behaviors do indeed match. While formal modelling techniques abound (Unified Modeling Language (UML), [1], Specification and Description Language (SDL)[2], Use Case Maps (UCM)[3]), testing is often performed with ad hoc coded tests using frameworks such as JUnit [5]. There is very little code reuse between tests and displaying the results often accounts for 50% of the code written to define tests.

Formal models frequently use Message Sequence Charts (MSCs) [4] (Figure 1) (Pragmadev studio) to enable the software engineer to visualize the behavior of a system even before it has been implemented giving them the possibility to detect design flaws early and thus avoid costly testing iterations [6][7].

The formal test specification language Testing and Test Control Notation (TTCN-3) [8] provides advantages over frameworks like Junit, with strong typing, a powerful matching mechanism, and a separation of concerns between the abstract test specification layer and the concrete layer that handles coding/decoding data which can result in significant code reuse [16].


PragmaDev Studio - MSC Diagrams
Diagram Edit Search View Export Windows Help
MSC_1
entity_A
entity_B
entity_C
stimuli_1
stimuli_2
response_1
response_2
Figure 1. basic MSC

Especially interesting is the support of MSCs to display test results that is provided by commercially available TTCN-3 execution tools like TTworkbench, [9], Testcast [10], PragmaDev Studio [11], Titan [12]. All of these tools use MSCs to display test results which is especially efficient when the system is composed of multiple components that interact with each other as shown in Figure 2.

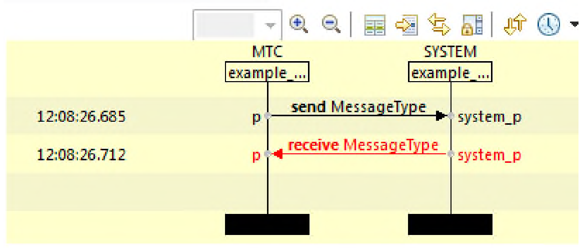
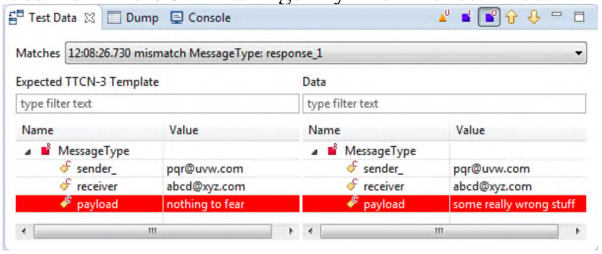


Figure 2. Test results as MSC

However, all of these tools are confronted with the same problem of displaying complex structured data in the limited space provided by MSCs. Thus, they avoid the display problem altogether by showing only the data type of the message (Figure 2 shows TTworkbench) and show the content of the message in a separate table (Figure 3 for TTworkbench) when clicking on one of the arrows of the MSC. This requires a tedious message by message inspection of the MSC. However, this feature is critical in order to allow to spot errors efficiently. The TTworkbench tool is particularly interesting because it is the only one that shows the test oracle, the expected message against the data received from the SUT and flags any mismatches in red.



Matches 12:08:26.730 mismatch MessageType: response_1

Expected TTCN-3 Template

Name	Value
MessageType	
sender_	pqr@uvw.com
receiver	abcd@xyz.com
payload	nothing to fear

Data

Name	Value
MessageType	
sender_	pqr@uvw.com
receiver	abcd@xyz.com
payload	some really wrong stuff

Figure 3. Detailed message content display

II. TTCN-3 CONCEPT OF TEMPLATE

The central concept of TTCN-3 is the template language construct that enables describing both test stimuli and test oracles as structured data in a single template. This in turn is used by the TTCN-3 tools internal matching mechanism that compare the values of a template to the actual values contained in the response message both on message based and procedure based communication. More important is that the template has a precise name and is a building block that can be re-used using its name to specify the value of an individual field or another template that itself can be re-used by specifying a modification to its values. This is a concept of inheritance. For example, one may specify the templates for the sender and the receiver entities separately:

```
template charstring entityA_Template
                         := "abcd@xyz.com";
template charstring entityB_Template
                         := "pqr@uvw.com";
```

A stimuli message can then be specified as:

```
template MessageType stimuli_1 := {
    sender  := entityA_Template,
    receiver := entityB_Template,
    payload  :=  "it  was  a  dark  and
stormy night"
}
```

The response template can itself reuse the above entity addresses by merely reversing their roles (sender/receiver):

```
template MessageType response_1 := {
    sender := entityB_Template,
    receiver := entityA_Template,
    payload := "nothing to fear"
}
```

The TTCN-3 template modification language construct can be used to specify more stimuli or responses for the same pairs of communicating entities:

```
template MessageType stimuli_2
              modifies stimuli_1 := {
    payload := "the sun is shining at
last"
}
```

Templates can then be used either in send or receive statements to describe behaviors in the communication with the SUT. Such behavior can be sequential, alternative or even interleaved behavior. The TTCN-3 receive statement does more than just receive data in the sense of traditional general purpose languages (GPL). It compares the data received on a communication port with the content of the template specified. The following abstract specification means that upon sending template *stimuli_1* to the SUT, if we receive and match the response message to the template *response_1* we decide that the test has passed. Instead, if we receive and match *alt_response* we decide that the test has failed.

```
myPort.send(stimuli_1);
alt {
    [] myPort.receive(response_1){
       Setverdict(pass)
    }
    [] myPort.receive(alt_response){
       Setverdict(fail)
    }
}
```

III. SELECTING DATA FIELDS TO DISPLAY

While most of the tools provide test results in form of an XML file precisely for enabling users to use their own proprietary test results display methodology, instead, we decided to modify the tool's source code. The motivation for this approach was to avoid having to re-develop the MSC display software and especially the message selection mechanism that displays the detailed structured data table but also to maintain consistency between the abstract layer and the TTCN-3 tool. Thus, we preferred to modify the display software source code itself to display selected data so that the existing detailed data features when clicking on the arrows of the MSC are preserved and don't need to be re-developed. Our approach is a first in TTCN-3 tools.

The central concept of our approach is to use the standard TTCN-3 extension capabilities that can be specified at the abstract layer using the *with-statement* language construct. TTCN-3 extensions were devised in the TTCN-3 standard to precisely allow tools to handle various non-abstract aspects of a test such as associated codecs and display test results in the most appropriate way the user desires. While the language is standardized, there is no standardization on how a tool operates and, in particular, how it displays test results. Here, we use the template definition itself and its associated *with-statement* in the abstract layer as a way to specify the fields that will be displayed on the MSC during test execution since the template is used by the matching mechanism. In the following example, we are testing some database content for information about cities that is a well multi-layered data structure with fields and sub-fields as follows.

```
template CityResponseType response_1
                              := {
   location := {
      city := "ottawa",
      district := "ontario",
      country := "canada"
   },
   statistics := {
      population := 900000,
      average_temperature := 10.3,
      hasUniversity := true
   }
} with { extension "{display_fields
     { location {city},
       statistics { population }}}"; }
```

The above TTCN-3 *with-statement* uses the standard TTCN-3 *extension* keyword. It contains a user definition that is represented as a string. The content of this string is not covered by the TTCN-3 syntax but by syntax defined by the user. Thus, it is the responsibility of the user to handle syntax and semantic checking of that string's content. First, we have defined a keyword called *display_fields* to indicate that the specification is about selecting the fields to display. Then, we specify a list of fields and subfields to display. The curly brackets indicate the scope of subfields. For example, we specified that we want to see the *city* subfield of the *location* field and the *population* subfield of the *statistics* field. This hierarchy is necessary because various fields may have subfields with identical names.

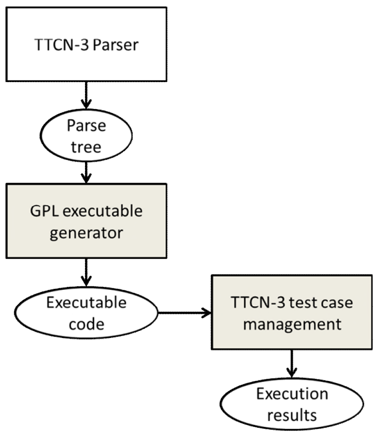


Figure 4. Structure of a TTCN-3 tool

We have implemented this feature on the Titan [12] open-source TTCN-3 execution tool software since this feature requires modifying the source code of the tool. None of the commercial TTCN-3 tool vendors make their source code available. Two areas of the Tool's source code (see Figure 4) were modified:

- the source code for the executable (GPL) code generator that will propagate the selected fields to display.
- the TTCN-3 test case management code that handles the MSC display

This did not require modification of the parser since the content of the *with-statement* is user defined, thus not modifying the grammar of the TTCN-3 language. However, the user definition turns up in the parse tree that is used for test execution code generation. It is during this code generation that we take into account this extension for the display specification. Most TTCN-3 execution software is based on execution code generated in a general purpose language (GPL) like Java for TTworkbench or C++ for Titan and PragmaDev studio and multiple strategies for TestCast. The general principle of these GPL generated code is to transform the abstract TTCN-3 definitions into executable GPL code, for example, in the TITAN tool, the abstract TTCN-3 template definition *response_1* shown previously becomes a series of C++ definitions, one for defining constants and the other to define the template matching mechanism as follows:

```
static const CHARSTRING cs_7(2, "75"),
cs_2(6, "canada"),
cs_8(6, "france"),
cs_4(8, "new york"),
cs_3(13, "new york city"),
cs_1(7, "ontario"),
cs_0(6, "ottawa"),
cs_6(5, "paris"),
…
```

The above definitions are in turn used to generate the C++ source code for the template definition as follows:

```
static void post_init_module()
{
TTCN_Location
current_location("../src/NewLoggingStudy
Struct.ttcn3", 0,
TTCN_Location::LOCATION_UNKNOWN,
"NewLoggingStudyStruct");
current_location.update_lineno(42);
#line 42
"../src/NewLoggingStudyStruct.ttcn3"
template_request__1.city() = cs_0;
template_request__1.district() = cs_1;
template_request__1.country() = cs_2;
current_location.update_lineno(48);
#line 48
"../src/NewLoggingStudyStruct.ttcn3"
{
LocationType_template& tmp_0 =
template_response__1.location();
tmp_0.city() = cs_0;
tmp_0.district() = cs_1;
tmp_0.country() = cs_2;
}
```

Thus, we had to use the same technique of C++ variable definitions to pass on our field display definitions since at run-time, the parse tree is no longer available. The test result MSC is considered as logging activity. Here this is illustrated by calling TITAN function *log_event_str()* that actually writes the template in the source code because this is the test oracle as follows:

```
alt_status
AtlasPortType_BASE::receive(const
CityRequestType_template&
value_template, CityRequestType
*value_ptr, const COMPONENT_template&
sender_template, COMPONENT *sender_ptr)
{
…
TTCN_Logger::log_event_str(": extension
{display_fields { location {city},
statistics { population, temperature}}}
@NewLoggingStudyStruct.CityRequestType :
"),
my_head->message_0->log(),
TTCN_Logger::end_event_log2str()),
msg_head_count+1);
…
```

Using the above source code, during the test execution, the Titan tool writes a log file that contains the matching mechanism results, i.e. the field names and instantiated values of the TTCN-3 template but also after the code modifications, the *display_fields* specifications as follows:

```
09:33:49.443373 Receive operation on
port atlasPort succeeded, message from
SUT(3): extension { display_fields {
location {city}, statistics {
population, temperature}}}
@NewLoggingStudy.CityResponseType : {
city := "ottawa", district := "ontario",
country := "canada", population :=
900000, average_temperature :=
10.300000, hasUniversity := true } id 1
```

The above data is used by the MSC display tool (on Eclipse) and shows two different kinds of information. The first is the content of our *display_fields* definition and the second is the full data that was received and matched. In fact all we had to do was to prepend the field selection logic to the actual log data that remained unchanged. The first will enable the MSC display software to display only the data requested like on Figure 9 while the second one is used for the detailed message content table that is obtained traditionally by clicking on the selected arrow of the MSC like on Figure 3.

While in open source Titan the execution code is written in C++, the actual Eclipse based MSC display is written in Java. Thus we had to modify the Java code that displays the MSC as well. Now, this is the implementation that is valid for Titan tool only. Each tool vendor has different coding approaches and would require different code generation strategies. Unfortunately since they do not make their source code available, all we can do is to strongly encourage these tool vendors to implement our MSC display approach.

IV. THE SIP PROTOCOL TESTING EXAMPLE

The SIP protocol (Session Initiation protocol) [13] is a very complex protocol using complex structured data including a substantial proportion of optional fields. The SIP protocol TTCN-3 test suites are available from ETSI [14] Traditional TTCN-3 tools will display all the fields in the detailed message content table. The user must click on some fields of interest to see the structured content. However, most real application messages make use of only a fraction of all the available fields. Thus, our approach can easily display this fraction of available fields in the MSC.

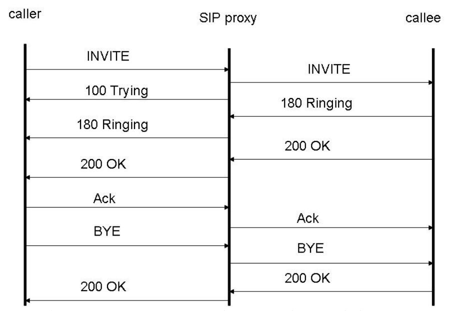


Figure 5. SIP protocol example model MSC

The ETSI definitions for the SIP protocol have used a strategy to try to alleviate the data type display problem in test result MSCs. The approach consists of redefining several times the same structured data type giving different names like in the following excerpt where there is a type for an INVITE method and the BYE request that are absolutely identical from a field definition point of view but they will display differently on the MSC using data types only:

```
type record INVITE_Request {
      RequestLine requestLine,
      MessageHeader msgHeader,
      MessageBody messageBody optional,
      Payload payload optional
}
type record BYE_Request {
      RequestLine requestLine,
      MessageHeader msgHeader,
      MessageBody messageBody optional,
      Payload payload optional
}
```

Where the main field is defined as:

```
type record RequestLine {
      Method method,
      SipUrl requestUri,
      charstring sipVersion
}
```

And the method type is an enumerated type:

```
type enumerated Method {
      ACK_E,
      BYE_E,
      CANCEL_E,
      INVITE_E,
      …
}
```

All of this can be used to specify a template that has all its fields set to any value except for the method as follows:

```
template INVITE_Request
            INVITE_Request_r_1 := {
   requestLine := {
      method := INVITE_E,
      requestUri := ?,
      sipVersion := SIP_NAME_VERSION },
   msgHeader := {
     callId := {
        fieldName := CALL_ID_E,
        callid := ?
     },
     contact := ?,
     cSeq := {
      fieldName := CSEQ_E,
      seqNumber := ?,
      method := "INVITE" },
      fromField := ?,
      toField := ?,
      …
}
```

We can select the field for the SIP method to display in the test results MSC by adding the *with-statement* to the above template as follows:

```
with { extension "{display_fields
     { requestLine { msgHeader {cSeq
{method} }} }}"; }
```

This will produce exactly the test results MSC that will be identical to the model MSC shown on Figure 5.

V. AN AVIONICS TESTING EXAMPLE

The whole idea of selecting data to display on a test results MSC originated specifically in an industrial application that we have worked on for testing the Esterline Flight Management System (FMS) [15]. The FMS shown on Figure 6 enables pilots to enter flight plans and display the flight plan on the FMS screen. A flight plan can be modified as a flight progresses. Flight plans and modifications are entered by typing the information using the alphanumeric key pad that consist of letters of the alphabet, numbers and function keys. For test automation purposes, key presses can be simulated by sending messages to a TCP/IP communication port. The content of a screen can be retrieved anytime with a special function invocation that will return a response message on the TCP/IP connection. Thus, we have the behavior of a typical telecommunication system sending and receiving messages with the difference that the response message must be requested explicitly, it is not coming back spontaneously and is subject to response delays that must be handled carefully in case of time outs.

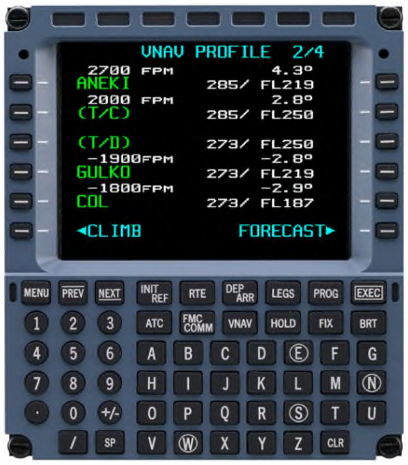


Figure 6. Flight Management System

In this case, stimuli messages are simple characters or names of function keys. These messages are by definition very short and can easily be displayed in full on the test results MSC. For such short messages, we have devised a default display option where if there is no with-statement with a display field specification for a given template, the MSC will display all data of this message. This is particularly optimal for short message content like the FMS key presses. The original test results MSC provided by Titan was displayed using useless message type names as shown on Figure 7 .

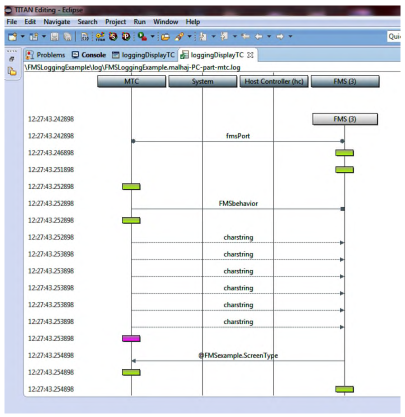


Figure 7 Original TITAN test results MSC display

It is clear from looking at Figure 7 that this MSC is not useful from an overview point of view while our approach on Figure 9 shows the messages values which allows the user to explore rapidly the test results before deciding to go for a fully detailed view of the results when for example the matching of the test oracle with the resulting response shows a failure. This is where the comparison with a model such as UCM is particularly easy to achieve as shown on Figure 8.

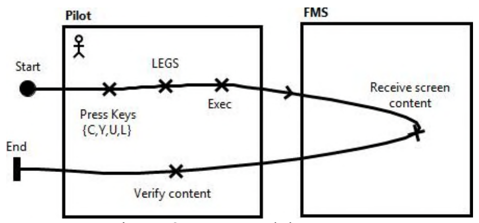


Figure 8. FMS model as UCM

The content of the screen is mapped to a data structure that contains fields for the various lines of the screen and also subfields to describe the left and the right of the screen. The FMS has 26 such fields, a title line, 6 lines structured into 4 subfields and a scratch pad line. Normally a test is designed to verify a given requirement which consists in verifying that a limited number of fields have changed their values. For example, the result of a sequence of stimuli may have changed the field that displays the destination airport on line 2 in the right part of the screen.

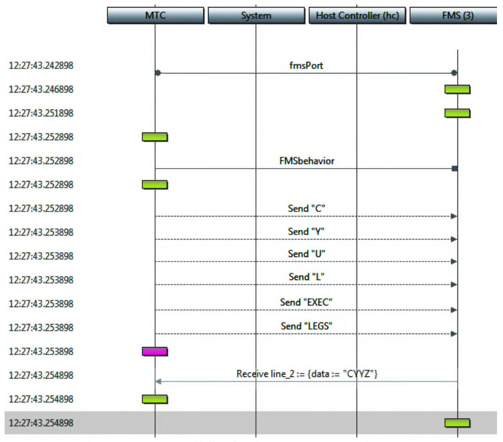


Figure 9. Modified Titan test result MSC

VI. CONCLUSION

In this research, we have shown that when using TTCN-3, it is an advantage to display selected information of complex structured data so as to have an overview on the test results and be able to locate an area of interest quickly and efficiently in test results.

ACKNOWLEDGMENT

We would like to thank CRIAQ, MITACS, ISONEO SOLUTIONS and CMC Esterline for their financial support on this project.

REFERENCES

[1] S. Jagadish, C. Lawrence and R.K. Shyamasunder, cmUML - A UML based Framework for Formal Specification of Concurrent, Reactive Systems, Journal of Object Technology (JOT), Vol. 7, No. 8, Novmeber-December 2008, pp 188-207.

[2] A. Ollsen, O. Færgemand and B. Møller-Pedersen, Systems Engineering using SDL 92, Elsevier Science B.V., Amsterdam, The Netherlands, 1994.

[3] R.J.A. Buhr and R. S. Casselman, Use Case Maps for Object-Oriented Systems, Prentice Hall Inc., Upper Saddle River, New Jersey, USA, 1995. ISBN:0-13-456542-8

[4] R. Alur, and M. Yannakakis, Model checking of message sequence charts, International Conference on Concurrency Theory. Springer Berlin Heidelberg, 1999, pp 114-129

[5] Y. Cheon, and G. T. Leavens, A simple and practical approach to unit testing: The JML and JUnit way. In European Conference on Object-Oriented Programming, June 2002, pp. 231-255. Springer Berlin Heidelberg.

[6] A. Miga, D. Amyot, F. Bordeleau, C. Cameron, and M. Woodside, Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. Tenth SDL Forum (SDL'01), Copenhagen, Denmark, June 2001.. LNCS 2078, 268-287

[7] J. Kealey, and D. Amyot, (2007) Enhanced Use Case Map Traversal Semantics. In: E. Gaudin, E. Najm, and R. Reed (Eds.): *13th SDL Forum (SDL 2007)*, Paris, France, September 2007. LNCS 4745, Springer, 133-149.

[8] ETSI ES 201 873-1 version 4.6.1 (2014-06) The Testing and Test Control Notation version 3 Part 1: TTCN-3 Core Language

[9] TTworkbench,Spirent, https://www.spirent.com/Products/TTworkbench

[10] Testcast, Elvior: http://www.elvior.com/testcast/introduction

[11] PragmaDev Studio, http://www.pragmadev.com/

[12] Titan, https://projects.eclipse.org/proposals/titan

[13] SIP RFC 3261, https://www.ietf.org/rfc/rfc3261.txt

[14] SIP TTCN-3, ETSI http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/27-publicts-sip

[15] FMS, href= http://www.esterline.com/avionicssystems/en-us/productsservices/aviation/navigationfmsgps/flightmanagementsystems.aspx

[16] B. Stepien, L.Peyton, M. Shang and T.Vassiliou-Gioles, "An Integrated TTCN-3 Test Framework Architecture for Interconnected Object-based Internet Applications", International Journal of Electronic Business, Inderscience Publishers, Vol. 11, No. 1, pp. 1-23, 2014. DOI: http://dx.doi.org/10.1504/IJEB.2014.057898