Model-based Visualization of Execution Traces 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— **Support for model-based visualization of execution traces in testing tools is limited at best, even though model-based approaches to specifying and visualizing behavior are well known and commonly used in the development of software applications. There has been active research on generating test scripts from formal models of behavior, but most testing tools in industry have little or no support for structuring test results based on behavior models. We present an approach for extending the Testing and Test Control Notation version 3 (TTCN-3) test results message sequence chart feature to address this problem. TTCN-3 is a test specification and test implementation language owned by European Telecommunications Standards Institute (ETSI). This leverages TTCN-3 support of the *with-statement* language construct to allow for custom configuration of the test results display. The approach is illustrated with two examples: testing a communication protocol used for controlling multimedia sessions called Session Initiation Protocol (SIP) and testing an avionics flight management system.**

Keywords-Software modeling; Behavior modeling; Software testing; Automated Testimg; TTCN-3.

I. INTRODUCTION

This paper extends, updates, and provides more detail on earlier research results presented at the International Conference on Trends and Advances in Software Engineering [1].

Support for model-based visualization of execution traces in testing tools is limited at best, even though model-based approaches to specifying and visualizing behavior are well known and commonly used in the development of software applications. There has been active research on generating test scripts from formal models of behavior, but most testing tools in industry have little or no support for structuring test results based on behavior models.

We present an approach for extending the TTCN-3 test results message sequence chart feature to address this problem. This leverages TTCN-3 support of the *with-statement* language construct to allow for custom configuration of the test results display. TTCN-3 is a test specification and test implementation language created by industry and academia experts at the European Telecommunications Standards Institute (ETSI) [2]. TTCN-3 is a powerful scripting language that is employed to test web applications [3], composite applications enabled in SOA [4]. It also has been extended to web penetration testing that involves- SQL injection and XSS attacks [5]. Using separation of concerns modeling principle, TTCN-3 separates the Abstract Test Suite (ATS) from the coding/decoding and communication, and presentation details; providing powerful matching mechanism that separates behavior and the conditions governing the behaviors and there by promoting systematic approach to testing. This separation of concern between ATS and Adapter Test Layer provides full portability of test suites, making them independent of platform implementation [6].

The approach is illustrated with two examples: testing a SIP protocol and testing an avionics flight management system.

Model-based specification of behavior while developing software applications can be done using interaction diagrams in the Unified Modeling Language (UML) [7], message sequence charts (MSC) in the Specification and Description Language (SDL) [8] and use case maps (UCM) in the User Requirements Notation (URN) [9]. The relationship between model-based specifications of behavior and testing is well understood [10]. A UML Testing Profile has been developed to support model driven testing [11] that has been mapped to test languages like Junit [12] and TTCN-3 [13]. Automatic generation of test scripts from models has been an active area of research using UML interaction diagrams [14], UCM [15] and MSC [16].

Figure 1 shows an example of a simple MSC diagram using Pragmadev Studio [17]. Such a diagram enables 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 with model checking [18]. MSC diagrams and UML interaction diagrams are similar, and MSC diagrams can be derived from UCM diagrams [19] [20]. MSC diagrams have been used to address security [21], conformance [22], performance [8] and business processes [9] as well as concurrency and real-time processing [7].

Testing tools that are currently available on the market do not adequately support for structuring test results in relationship to model-based specifications. Test frameworks, like Junit, which are oriented towards unit tests, have no built-in support at all for MSC or similar diagrams. Formally modelled test frameworks, like TTCN-3, are oriented towards integrated component-based system testing and do have basic support for message sequence charts. Unfortunately, the available support is not sufficient when

working with complex, structured data. TTCN3; however, does provide 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 [23].

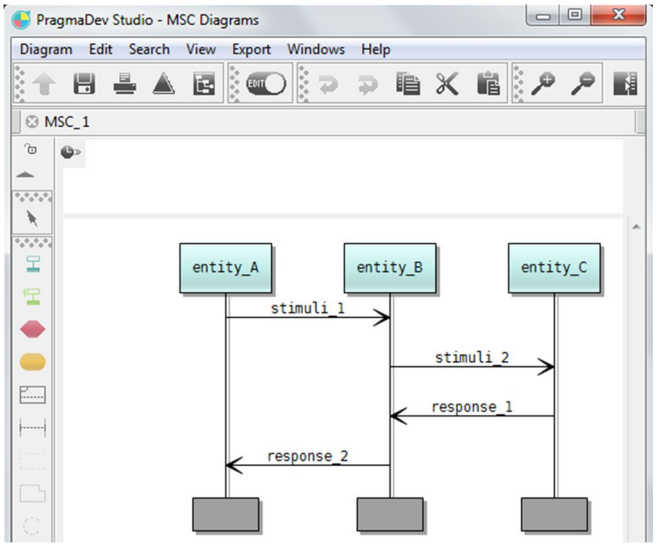

Figure 1. Simple MSC

Figure 2 shows the test results for a particular test script in the TTWorkbench tool from Spirent [24]. It is displayed in the context of an MSC diagram. There is similar support for displaying results in the context of an MSC diagram in all the industry tools that support the TTCN-3 standard, including Testcast from Elvior [25], Tester from PragmaDev [17] and the open source tool Titan that was originally developed at Ericsson [26] [26] before being made available as an Eclipse project. However, the TTworkbench tool is significant because it is the only one that compares the test oracle (the expected response message) against the data received from the system under test (SUT) and flags any mismatches in red.

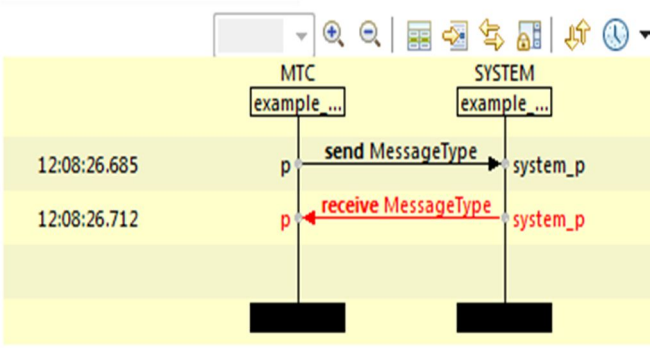

Figure 2. Test results as MSC

While all of these tools are able to display test results in the context of a basic MSC diagram, it is of limited use for visualizing, analyzing and navigating test results in a productive fashion. Typically, the "MessageType" shown in Figure 2 is complex structured data. Displaying only the type of the structured data does not really provide useful information for understanding what has gone wrong.

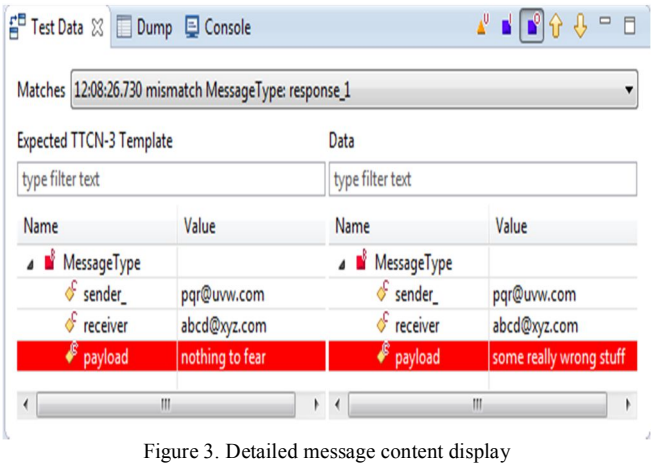

Figure 3. Detailed message content display

To see the actual data, the user has to click on one of the arrows in the diagram and the content or value of the message is shown in a separate window (Figure 3 above). It is difficult to understand the error, without being able to see the step by step details of the data involved in each message that leads to the error. This requires a tedious message by message inspection for each arrow in the MSC diagram. On the other hand, there is too much data in a complex data type to display all the data for each message arrow in Figure 2.

Note that a model MSC and a test result MSC may not be identical. A model MSC may contain alternate behavior as shown in Figure 4, while the test result MSC is by definition only a trace through the model MSC that would traverse only one of the branches of the alternative behavior.

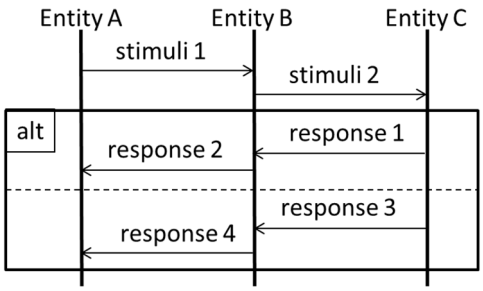

Figure 4. Complex Model MSC

We would like to address this MSC visualization problem by enabling custom configuration of the MSC diagram so that only a specified subset of the data will be displayed in order to provide the tester with an overview of the test results and the flow of data from message to message. That way, only the most critical messages need to be clicked on to view the full details in a separate window. To achieve this goal we found that we can use the TTCN-3 standard extension mechanism, which allows the tester to

give instructions to the execution tools without having to change the syntax or the semantics of the TTCN-3 language itself.

The paper is organized as follows: Section II presents the TTCN-3 concept of template; Section III presents the Selecting data fields to display in TTCN-3; Section IV presents two application examples: testing SIP protocol and testing an avionics flight management system; and section V presents the conclusion.

II. TTCN-3 CONCEPT OF TEMPLATE

The central concept of TTCN-3 is the template language construct that enables the specification of both test stimuli and test oracles as structured data in a single template. This, in turn, is used by the TTCN-3 built-in matching mechanism to compare the values of a template to the actual values contained in the response message. This is supported for both message-based and procedure-based communication. More importantly, the template has a precise name and is a building block that can be re-used to specify the value of an individual field, or it can be re-used by another template that specifies a modification to its values. This is a concept of inheritance.

TTCN-3 has a data typing capability mostly because early applications of TTCN-3 were in the telecommunication sector where data typing is common in order that various test events could share parts of data. In this case, these data types would be abstract, independent from any coding/decoding considerations. Abstract data has the advantage of enabling generic matching mechanisms that are applicable to any kind of application. For example, splitting data into fields of a structured data type enables easy referencing to a specific field for all sorts of manipulations.

The data string: "abcd10xyz" could be split into 3 fields of a structured data type with its use after decoding into a variable:

```
type record MyType
{
 charstring field_1,
 integer field_2,
 charstring field_3
}

var MyType myVar :=
{
 field_1 := "abcd",
 field_2 := 10,
 field_3 := "xyz"
};
```

This approach is more efficient than, for example, an assertion statement used in JUnit such as:

```
assertTrue(substring(response_string,
0,4).equals("abcd");
```

An example, to illustrate re-usability of template consists in specifying 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 by re-using them 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 the roles of (sender and 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 and can make use of timers to check for lost messages. The TTCN-3 *receive* statement does more than just receiving 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 instead the *alt_response* template we decide that the test has failed and finally if the timer expires we decide that the test is inconclusive.

```
Timer myTimer(5.0);
myPort.send(stimuli_1);
alt
{
   [] myPort.receive(response_1){
```

```
        setverdict(pass)}
   [] myPort.receive(alt_response){
        setverdict(fail)}
   [] myTimer.timeout {
        Setverdict(inconc)}
}
```

The use of structured data types for describing message content is not new as we already mentioned, but their internal representation in a generic way has the advantage to allow a generic matching mechanism. In other words, instead of specifying multiple assertions, all the fields of the template are checked at once without any additional effort from the test designer.

III. SELECTING DATA FIELDS TO DISPLAY

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 included in the TTCN-3 standard to 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.

Most of the TTCN-3 tools provide test results in the form of an XML file. This enables users to customize their own proprietary test results display and to store test results in a file for later consultation. We wanted to avoid having to re-develop the MSC display software and especially the message selection mechanism that displays the detailed structured data table. We also wanted to maintain consistency between the abstract and concrete layers for the TTCN-3 tool. As a result, we decided to modify the TTCN-3 test execution source code to handle the extensions we specified using the *with-statement* language construct. This approach is a first in TTCN-3 tools. We updated the display software source code to display data values as configured by the user using the *with-statement* language construct. This ensured that the existing detailed data features when clicking on the arrows of the MSC were preserved.

Here, we use the template definition itself and its associated *with-statement* in the abstract layer as a way to specify the field values that will be displayed in the MSC diagram during test execution since the template is used by the matching mechanism. The grammar in Bachus Naur Form (BNF) for the TTCN-3 *with-statement* is as follows:

```
455.WithStatement::=WithKeyword
WithAttribList

456.WithKeyword ::= "with"

457.WithAttribList::="{"
MultiWithAttrib "}"

458.MultiWithAttrib::=
{SingleWithAttrib [SemiColon]}

459.SingleWithAttrib              ::=
AttribKeyword       [OverrideKeyword]
[AttribQualifier]FreeText

460.AttribKeyword::=EncodeKeyword
|VariantKeyword|DisplayKeyword
|ExtensionKeyword|OptionalKeyword

461.EncodeKeyword ::= "encode"

462.VariantKeyword ::= "variant"

463.DisplayKeyword ::= "display"

464.ExtensionKeyword ::= "extension"

465.OverrideKeyword ::= "override"

466.AttribQualifier      ::=      "("
DefOrFieldRefList ")"

467.DefOrFieldRefList             ::=
DefOrFieldRef {"," DefOrFieldRef}

468.DefOrFieldRef                 ::=
QualifiedIdentifier |((FieldReference
|        "["        Minus        "]")
[ExtendedFieldReference]) |AllRef

469.QualifiedIdentifier           ::=
{Identifier Dot} Identifier

470.AllRef      ::=      (GroupKeyword
AllKeyword     [ExceptKeyword     "{"
QualifiedIdentifierList"}"])        |
((TypeDefKeyword      |TemplateKeyword
|ConstKeyword          |AltstepKeyword
|TestcaseKeyword      |FunctionKeyword
|SignatureKeyword   |ModuleParKeyword)
AllKeyword           [ExceptKeyword"{"
IdentifierList"}"])
```

In the above BNF, we can observe the definition of the *DisplayKeyword* on line 463. Unfortunately, this display construct cannot be used for our purposes. Effectively, what is meant by display is the way a given identifier is displayed. The example below given in the standard is very clear means that the type name “MyService” will be displayed in the test execution results log as “ServiceCall”.

```
type record MyService
{
   integer i,
   float f
```

```
}
  with { display "ServiceCall" }
```

In the following example, where we are testing some database content for information about cities that is a well multi-layered data structure with fields and sub-fields, we have used the extension keyword defined in line 464 of the above BNF 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, within this string, we have defined a keyword called *display_fields* to indicate that the extension specification is about selecting the fields to display. Then, we specify a list of fields and subfields of the data type being used to display. The curly brackets indicate the scope of subfields. In the above 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.

We have implemented this feature on the Titan [26] 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 5) were modified:

- The source code for the GPL executable code generator that will propagate the selected fields to display while generating execution logs.
- The TTCN-3 test case management code that generates the Execution results log used by the MSC display software.

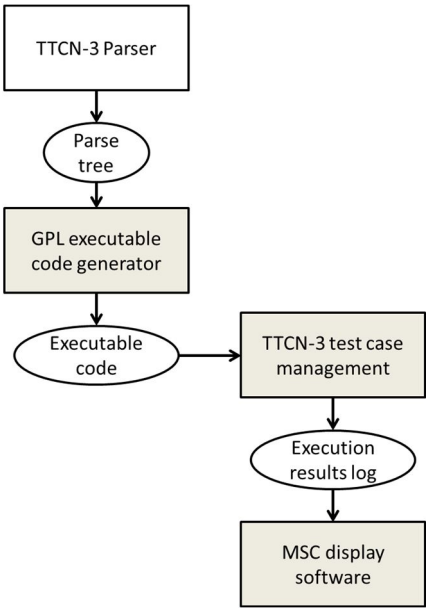

Figure 5. Structure of a TTCN-3 tool

This did not require modification of the TTCN-3 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 test execution 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 the 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 where, for example, the city field gets assigned the cs_0 constant that represents the string "ottawa":

```
static void post_init_module()
{
TTCN_Location
current_location("../src/NewLoggingSt
udyStruct.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;
}
```

We use the same technique of C++ variable definitions to pass on the value of our field display definitions since at run-time, the parse tree is no longer available. Test results are written in a log file. The TTCN-3 MSC feature reads that same log file to build and display the MSC itself. Here this is illustrated by calling TITAN function *log_event_str()* using the string value of the *display_fields* extension as defined in the template being used 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,
average_temperature}}}
@NewLoggingStudyStruct.CityRequestTyp
e : "),
my_head->message_0->log(),
TTCN_Logger::end_event_log2str()),
msg_head_count+1);
…
```

In that generated source code, only the *display_fields* string and the data type are shown as strings. The content of the message itself is found in the variable *my_head->message_0*. Here the *log()* method will actually write all the fields with name and value in the log file.

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 (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 extract the requested fields data and to display it as shown in Figure 13 (at the end of this paper), 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 diagram as shown in Figure 3.

In the open source Titan tool, the execution code is written in C++, but actual Eclipse-based MSC display is written in Java. Thus we had to modify the Java code that displays the MSC as well.

It should be noted that this implementation is valid for the 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. APPLICATION EXAMPLES

We have worked on two examples, both drawn from industrial applications we were involved with. The first

example is a widely used abstract test suite for the SIP protocol that has a very complex structured data type. It illustrates the benefits of our MSC display approach because in this case, it is obvious that it is totally impossible to display all the fields of a SIP message, especially since most of them are optional and would contain no values. The second example is an avionics application that illustrates the overview qualities of our approach when trying to navigate through long sequences of test events. It consists of long sequences of key strokes and screen display results verification.

A. The SIP protocol testing example

The SIP protocol [27] is a very complex text based protocol. For example, an INVITE method message text would be as follows:

```
INVITE sip:user:passwd@127.0.0.1:5060
SIP/2.0
Call-ID: 121231231
Contact:  <sip:auser@127.0.0.1:5060>
Content-Length: 0
CSeq: 666 INVITE
From: "aDisplayName"
<sip:auser@127.0.0.1:5060>
Max-Forwards: 70
To: "aDisplayName"
<sip:user@127.0.0.1:5060>
Via: SIP/2.0/udp 127.0.0.1:5060
```

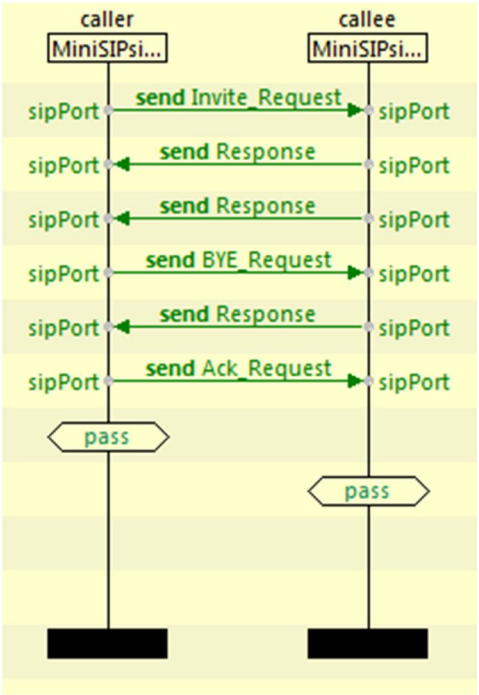

Figure 6. TTCN-3 generated MSC

When testing with TTCN-3, this text message is decoded and assigned to a complex structured data type including a substantial proportion of optional fields. The SIP protocol TTCN-3 test suites are available from ETSI [2]. The resulting MSC diagram would indicate only the message types but with no values as shown in Figure 6.

Name	Value
INVITE_Request	
requestLine	
method	INVITE_E
requestUri	
scheme	sip
userInfo	
userOrTelephoneSubscriber	user
password	passwd
hostPort	
host	127.0.0.1
portField	5060
urlParameters	omit
headers	omit
sipVersion	SIP/2.0
msgHeader	
accept	omit
acceptEncoding	omit
acceptLanguage	omit
alertInfo	omit
allow	omit
authenticationInfo	omit
authorization	omit
callId	
fieldName	CALL_ID_E
callid	121231231
callInfo	omit

Figure 7. Portion of the detailed message content inspection window

Traditional TTCN-3 tools will display all the fields in the detailed message content table, but the large amount of fields renders its inspection tedious. Relevant information may not be contiguous and requires scrolling through several pages of the message content table as shown in Figure 7. 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.

SIP application engineers actually use MSCs as models to guide development as shown in Figure 8 for a typical SIP call establishment and tear down. Note the alternative behavior portion of the MSC diagram, “alt”, that expresses the fact that the 100 TRYING event is optional. It could happen or not depending on what is the load of the system. It is mainly used to prevent premature timeouts.

Industrial applications we have worked on consisted in several hundreds of messages. The experience of walking through the messages content made us aware of the need for the approach we are suggesting. Thus here, there are no additional activities required to produce the MSC from the model and a straightforward comparison with the test results MSC can be performed.

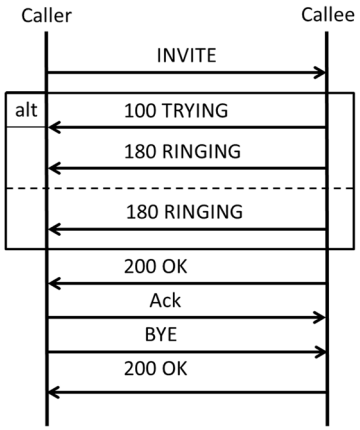

Figure 8. SIP protocol example model MSC

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 type 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 type names only. However, this approach has the disadvantage to hide various other active fields that differentiate the sequences of SIP events that otherwise would look the same.

```
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 these 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* field 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}
}} }}";
}
```

The resulting test execution MSC using our approach would look like Figure 9 which is easier to recollect with the model shown on Figure 8.

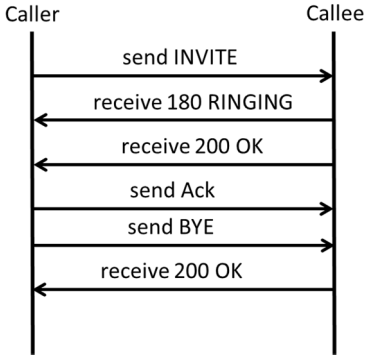

Figure 9. SIP test execution MSC

This approach is particularly effective for SIP response messages because they show the return code and the reason verb in the status line. Here, the SIP test suite designers have not used the type renaming strategy as they did for SIP requests. Thus, all responses will show the *Response* type name only on the traditional MSC. The status line is defined as follows:

```
type record StatusLine {
   charstring sipVersion,
   integer statusCode,
   charstring reasonPhrase
}
```

Which is used in the definition of the response type:

```
type record Response {
   StatusLine statusLine,
   MessageHeader msgHeader,
   MessageBody messageBody optional,
   Payload payload optional
}
```

A typical response message template can then use a *with-statement* that would only specify the *statusCode* and the *reasonPhrase* fields to be displayed.

```
template (value) Response
Response_200_s_1(
   CallId loc_CallId,
   CSeq loc_CSeq,
   From loc_From,
   To loc_To,
   Via loc_Via) := {
   statusLine := {
      sipVersion := SIP_NAME_VERSION,
      statusCode := 200,
      reasonPhrase := "OK"
      },
   msgHeader := {
         callId := loc_CallId,
  …
  }
with extension
"{display_fields
    { statusLine
   { statusCode,reasonPhrase}}";
```

This will produce exactly the test results MSC that will be a trace through the model MSC shown in Figure 8. Here the additional benefit would consist in declaring a single SIP message definition for SIP requests that would be used by any of the SIP message kinds as follow:

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

B. *An Avionics testing example*

The idea of selecting data to display on a test results MSC originated in an industrial application that we have worked on for testing the CMC Esterline Flight Management System (FMS) [28]. The FMS shown in Figure 10 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.

Figure 10. Flight Management System

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 via a screen query message. It is not coming back spontaneously and is subject to response delays that must be handled carefully in case of time outs.

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 in Figure 11.

The TTCN-3 abstract test suite has a data type definition for the content of the FMS screen that is returned as a response to the tester. The definition has been simplified due to confidentiality requirements from the industrial partner but this example renders the structural elements. Each line of the screen is defined using the *LineType* data type that has two subfields, the *title* and the *data*. Then, we define a screen type that is composed of two lines. The real application has a total of 26 subfields and illustrates well the fact that the entire screen content cannot be displayed on the test execution results MSC.

```
type record LineType {
      charstring title,
      charstring data
}

type record ScreenType {
      LineType line_1,
      LineType line_2
}
```

Using these data type definitions, we can define a template to match against the screen content and specify in the *with-statement* that we want to display only the data subfield of the second line:

```
    template ScreenType
screen_response_1 := {
        line_1 := {
            title := "waypoint 1",
            data := "CYUL"},
        line_2 := {
            title := "waypoint 2",
            data := "CYYZ"}
    }
      with {extension "display_fields {
                  line_2 { data }}"};
```

All of these being used in the test case behavior description as follows:

```
testcase loggingDisplayTC()
  runs on MTCType system SystemType {

    var SUTType sut :=
          SUTType.create("FMS");

    connect(mtc:fmsPort, sut:fmsPort);

    sut.start(FMSbehavior());

    fmsPort.send("C");
    fmsPort.send("Y");
    fmsPort.send("U");
    fmsPort.send("L");
    fmsPort.send("EXEC");
    fmsPort.send("LEGS");

    alt {
          []fmsPort.receive(
                screen_response_1) {
                setverdict(pass);
          }
          [] fmsPort.receive {
                setverdict(fail);
          }
    }

    sut.stop;
    setverdict(pass);
}
```

It is clear from looking at Figure 11 that this MSC is not useful as an overview because it shows only the same message type name for each stimulus while our approach in Figure 14 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. These values can be compared to those shown in a model such as the UCM diagram in Figure 12.

The UCM diagram of Figure 12 can be transformed into an MSC as described in [19] and shown in Figure 13. The MSC diagram in Figure 13 can then be compared with the test execution results shown in Figure 14.

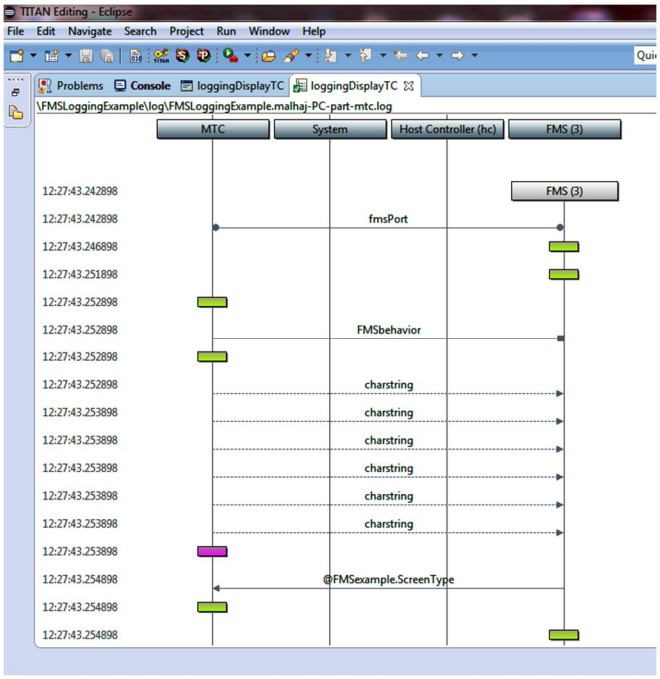

Figure 11. Original TITAN test results MSC display

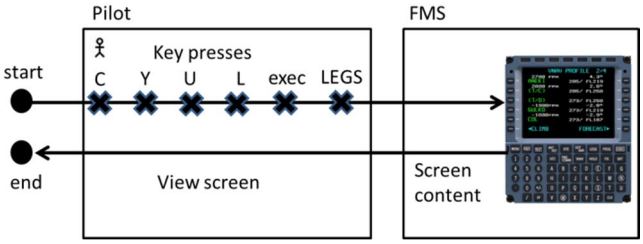

Figure 12. FMS model as UCM

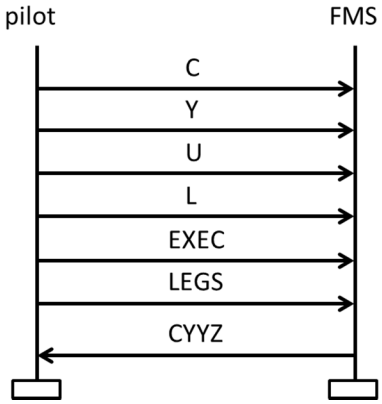

Figure 13. MSC diagram generated from UCM

The response message contains the content of the screen of the FMS. It 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 sides 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. This is specified as a display fields request to show only the *line_2* field and the subfield *data*.

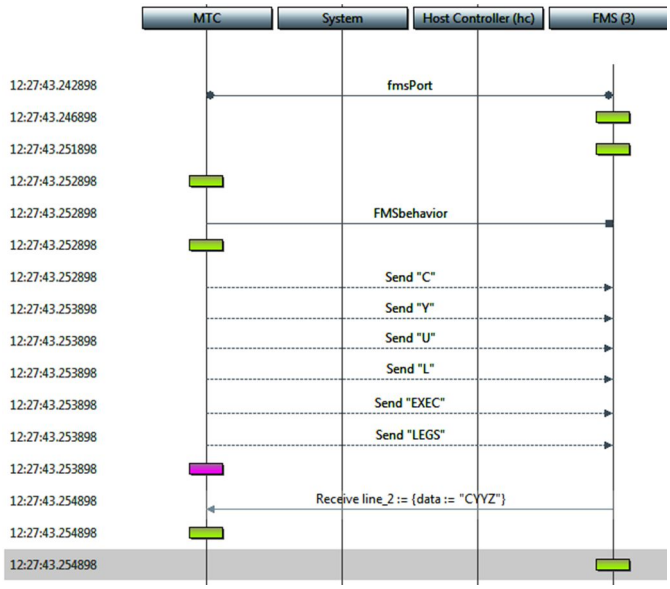

Figure 14. Modified Titan test result MSC

V. CONCLUSION

TTCN-3 tools provide limited support for visualizing test results in the context of MSC diagrams. We have shown how the with-statement language construct in TTCN-3 can be used to flexibly configure the display of data in the context of an MSC diagram without requiring changes to the TTCN-3 parser or grammar. The approach was validated by implementing it in the open-source Titan framework for TTCN-3 and applying it to two real-world examples: a SIP protocol and an avionics flight management system. Our approach successfully provided testers with a better mechanism for visualizing and navigating the complete set of test results in an efficient and effective manner.

ACKNOWLEDGMENT

We would like to thank CRIAQ, MITACS, Isoneo Solutions and CMC Esterline for their financial support of this research.

REFERENCES

[1] B. Stepien, M. Alhaj, and L. Peyton, "Visualizing Execution Models and Testing Results", 3rd Int'l Conference on Trends and Advances in Software Engineering (SOFTENG 2017), Venice, Italy, April 2017

[2] SIP TTCN-3, ETSI (2017) http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/27-publicts-sip

[3] Bernard Stepien, L. P. (2014, 06). Innovation and evolution in integrated web application testing with TTCN-3. International Journal on Software Tools for Technology Transfer, pp. 269-283.

[4] Peyton, L., Stepien, B., & Seguin, P. (2008). Integration Testing of Composite Applications. Waikoloa, HI: IEEE.

[5] Stepien, B., Xiong, P., & Peyton, L. (2011). A Systematic Approach to Web Application Penetration Testing Using TTCN-3. MCETECH (pp. pp. 1–16). Berlin Heidelberg: Springer-Verlag.

[6] Stepien, B. (2015, February 14). Testing and Test Control Notation. Retrieved from TTCN-3 in a Nutshell (2017): http://www.site.uottawa.ca/~bernard/ttcn3_in_a_nutshell.html

[7] 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, pp 188-207, November-December 2008.

[8] A. Mitschele-Thiel, and B. Müller-Clostermann, "Performance engineering of SDL/MSC systems", Computer Networks, 31(17), 1801-1815, 1999.

[9] A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and AJ Forster, "Business Process Management with the User Requirements Notation", Electronic Commerce Research, Springer, Vol. 9 No. 4, pp 269-316, 2009.

[10] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model - based testing approaches" Software Testing, Verification and Reliability 22.5 (2012): 297-312

[11] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams, "Model-Driven Testing Using the UML Testing Profile", Springer ISBN 978-3-540-72562-6 Springer Berlin Heidelberg New York, Springer-Verlag Berlin Heidelberg, 2008.

[12] 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, pp. 231-255. Springer Berlin Heidelberg, June 2002.

[13] ETSI ES 201 873-1 version 4.9.1, The Testing and Test Control Notation version 3 Part 1: TTCN-3 Core Language, last accessed August 2017: http://www.ttcn-3.org/index.php/downloads/standards

[14] E. G. Cartaxo, F. G. Neto, and P. D. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems", In Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on (pp. 1292-1297).

[15] D. Amyot, L. Logrippo, and M. Weiss, "Generation of test purposes from Use Case Maps", Computer Networks, 49(5), 643-660

[16] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe, SDL and MSC based test generation for distributed test architectures. In SDL Forum, Vol. 99, pp. 389-404, 1999, June.

[17] PragmaDev, last accessed August, 2017 at http://www.pragmadev.com/

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

[19] 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, LNCS 2078, 268-287, June 2001.

[20] J. Kealey and D. Amyot, "Enhanced Use Case Map Traversal Semantics", *13th SDL Forum (SDL 2007)*, Paris, France, LNCS 4745, Springer, 133-149, September 2007.

[21] B. Stepien, L. Peyton, and P. Xiong, "Using TTCN-3 as a Modeling Language for Web Penetration Testing", Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT 2012), Athens, Greece, IEEE Explore, pp 674 - 681 March 2012.

[22] H. Dan and R. M. Hierons, "Conformance testing from message sequence charts", In Software Testing, Verification and Validation (ICST), IEEE Fourth International Conference, pp. 279-288, March 2011.

[23] 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

[24] TTworkbench, Spirent, last accessed August 2017 at https://www.spirent.com/Products/TTworkbench

[25] Testcast, Elvior, last accessed August 2017 at http://www.elvior.com/testcast/ttcn-3

[26] Titan, last accessed August 2017 at https://projects.eclipse.org/proposals/titan

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

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