ELVIOR INPUT TO TDL DISCUSSION

20.1.2012

Andres Kull

Elvior
andres.kull@elvior.com

1 Introduction

This document is Elvior position about the Test Description Language (TDL).

The document gives an overview about the history of scenario-based modeling notations. The high-level requirements for TDL are proposed. Then is proposed which elements of the UML SD and TTCN-3 GFT should be reused to define TDL.

2 History of scenario-based notations

2.1 MSC

Message Sequence Chart (MSC) is an interaction diagram between communicating entities that exchange events. There have been several MSC versions standardized by ITU. The first version of the MSC standard was released in 1992.

The 1996 version added references, ordering and inline expressions concepts, and introduced HMSC (**H**igh-level **M**essage **S**equence **C**harts), which are the MSC way of expressing state diagrams.

The latest MSC 2000 version added object orientation, refined the use of data and time in diagrams, and added the concept of remote method calls.

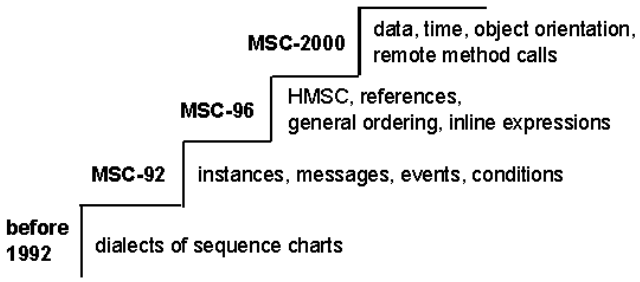


Figure 1: History of MSC

MSC 2000 is easy to use and well-formalized notation. It became popular especially in telecommunications industry. MSC has been used for requirements specification, systems design and test design.

MSC 2000 lost ground by the introduction of UML.

2.2 TTCN-3 GFT

TTCN-3 is a test dedicated high-level programming language for automated test scripts. Inspired in the popularity of the MSC the TTCN-3 graphical notation GFT was worked out. GFT became a formal graphical programming language for test cases by adding lot of TTCN-3 constructs to MSC-2000. As such it lost some of the important MSC benefits becoming too close to the programming language of describing the tester behaviour. For the people who can write TTCN-3 scripts it became useless because they prefer textual notation to graphical one. GFT also didn't

help people who don't know TTCN-3 because using GFT you have to know TTCN-3. Therefore currently GFT is used mainly for documenting purposes by rendering TTCN-3 textual test cases into GFT.

2.3 UML

UML standardization took over the results achieved on the MSC field. UML SD (Sequence Diagram) became the new MSC notation. Most of the MSC-2000 features were taken over and some new features were added. UML quickly became an industry standard and tool vendors updated their tools to support UML. It can be said that this was the end of original MSC-2000 and the life of MSCs were continued as UML SD.
Concepts of GFT were pushed to UML standards in form of UML TP (Test Profile). UML Test Profile provides the concepts to design test architecture, to define test data, and to define test behaviour. The concepts for defining the test behaviour are quite close to the respective concepts in GFT.

3 Why do we need TDL?

It can be asked why do we need another scenario-based notation for test definition (TDL) if we have already notations like GFT, UML SD and UML TP. The answer is that with GFT and UML TP one can specify the algorithm that the test component has to implement in order to test the SUT. Those notations provide means to graphically program the test cases from the test component point of view. This is not what the test engineer wants. The test engineer is often not very skilful in programming but they can define the test cases as message scenarios between the SUT instances and test components. They can define also that some fields in the messages have to be matched to the expected values. TDL should be the notation to define the test scenarios on the message sequence chart level without having to define how the test components should be implemented to achieve this. The difference between TDL and GFT/UML TP is "what" vs "how". UML SD is a good notation that TDL can be based on. TDL do not need everything that is present in UML SD and it might need something test specific to be added. In overall UML SD is in the notation on the same abstraction level than TDL is supposed to have.

4 TDL requirements

1. TDL must define the test case scenario without defining the execution algorithm of the test components explicitly. The abstraction level of TDL must be higher than executable tests.
2. TDL must define expected test scenarios as interaction between SUT and SUT components.
    a. Everything in actual scenario that do not match TDL scenario is a test failure.
    b. Defining explicitly verdicts on the scenario is not needed
3. TDL must be usable for test engineers who cannot code scripts.
4. TDL must base on UML meta-model.
   Rationales:

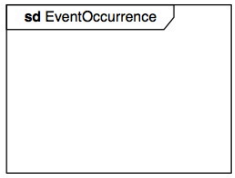
- UML is de-facto industry standard. It's not reasonable to compete with UML.
- All CASE tools support UML. Tool support for TDL acceptance is important.
- A good marketing strategy could be to create TDL as UML profile.

5. Any UML CASE tool should be possible to use for authoring TDL.
   Rationales:
   - At least for the beginning there are no special TDL authoring tools available.
6. TDL should reuse UML SD notation as much as needed and as less as possible.
   Rationales:
   - TDL should be easy to use notation therefore it should include from SD only the features that are mandatory for TDL. All nice-to-have features should be left out.
7. TDL must be formal for deriving executable test cases automatically from it.
8. TDL must have graphical presentation.
9. TDL may have tabular presentation.
10. TDL should have textual presentation.
11. TDL must be independent of test scripting language. Scenarios must use only data types and templates/instances references that can be defined in different module for the specific programming language in use.
    Rationales:
    - TDL should be possible to render into different scripting languages.
    - This will make market acceptance of TDL easier.
12. TDL must support associating data types and data instances to sequence charts.
13. Data types and instances of different programming language including UML must be supported.
14. Data types and instances must be defined in separate (language-specific) files.
15. Timing constraints must be modelled by defining min-max durations between events instead of using timer operations like start, stop, and timeout.
16. Multiple communicating SUT and test component instances must be supported.
17. TDL must support hierarchical composition of sequence charts similar to High-level Interaction Diagrams in UML and HMSCs in MSC-2000.
18. TDL must support asynchronous and synchronous (function call) messaging.
19. Test architecture definition must be supported – SUT and test components, ports, interface types.
20. TDL sequence charts must support context variables.
21. TDL must support attaching system requirements to the test scenarios.

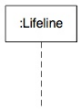
5 What to take over from UML SD?

The pictures and text about UML SD in the current chapter are copied from [1] and [2].

5.1 Frame

Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See 14.3.11, Interaction (from BasicInteraction, Fragments).

5.2 Lifeline

Lifeline		See 14.3.17, Lifeline (from BasicInteractions, Fragments).

Lifeline is a **named element,** which represents an **individual participant** in the interaction. While **parts** may have multiplicity greater than 1 then lifelines represent **only one** interacting entity.

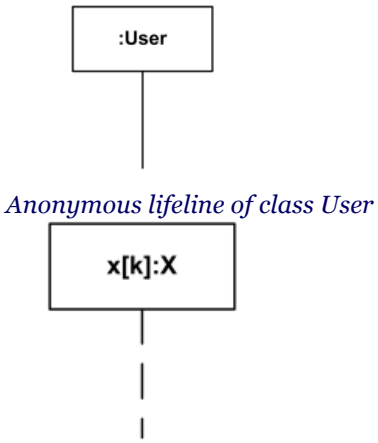
If the referenced connectable element is multivalued (i.e, has a multiplicity > 1), then the lifeline may have an expression (**selector**) that specifies which particular part is represented by this lifeline. If the selector is omitted, this means that an **arbitrary representative** of the multivalued connectable element is chosen.

A lifeline is shown using a symbol that consists of a rectangle forming its "head" followed by a vertical line (which may be dashed) that represents the lifetime of the participant.
Information identifying the lifeline is displayed inside the rectangle in the following format (slightly modified from what's in UML 2.4 standard):

lifeline-ident ::=
[***connectable-element-name*** ['[' ***selector*** ']']] [':' ***class-name***] [***decomposition***] | **'self'**
selector ::= ***expression***
decomposition ::= **'ref'** ***interaction-ident*** [**'strict'**]

where ***class-name*** is the type referenced by the represented connectable element.
The lifeline head has a shape that is based on the **classifier** for the part that this lifeline represents.

Lifeline *"data" of class Stock*



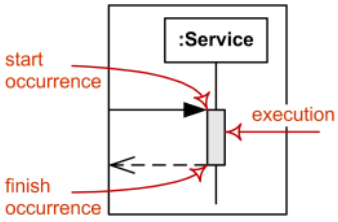
If the name is the keyword **self**, then the lifeline represents the object of the classifier that encloses the Interaction that **owns** the Lifeline.

5.3 Execution Specification

Execution specification, informally called **activation**, is **interaction fragment**, which represents a period in the participant's lifetime when it is

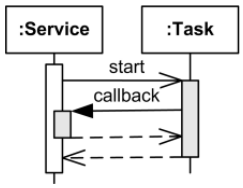
- executing a unit of behaviour or action within the **lifeline**,
- sending a signal to another participant,
- waiting for a reply message from another participant.

Note, that the **execution specification** includes the cases when behaviour is not active, but just waiting for reply. The **duration** of an execution is represented by two **execution occurrences** - the **start** occurrence and the **finish** occurrence.
Execution is represented as a thin grey or white rectangle on the lifeline.



Execution specification can be represented by a wider labeled rectangle, where the label usually identifies the action that was executed.

Overlapping **execution specifications** on the same lifeline are represented by overlapping rectangles.



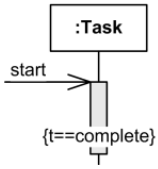
5.4 State Invariant

A **state invariant** is an **interaction fragment,** which represents a runtime **constraint** on the participants of the interaction. It may be used to specify different kinds of constraints, such as values of attributes or variables, internal or external states, etc.

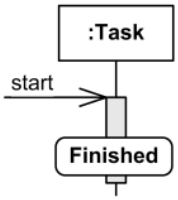
The constraint is evaluated immediately prior to the execution of the next occurrence specification such that all actions that are not explicitly modeled have been executed.

If the runtime constraint is true, the trace is a valid trace, otherwise the trace is an invalid trace and the test fails.

State invariant is usually shown as a constraint in curly braces on the lifeline.

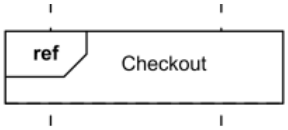


It could also be shown as a **state** symbol representing the equivalent of a constraint that checks the state of the object represented by the lifeline. This could be either the internal state of the classifier behaviour of the corresponding classifier or some external state based on a "black-box" view of the lifeline.



5.5 Interaction use

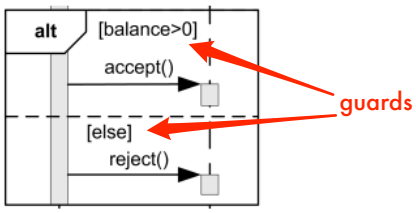
Interaction use is an **interaction fragment,** which allows to use (or call) another interaction. Large and complex sequence diagrams could be simplified with interaction uses. It is also common reusing some interaction between several other interactions.



5.6 Guard

A guard is a **constraint** used in interactions - a Boolean expression that **guards** an operand in a **combined fragment**.

An interaction constraint is shown in square brackets covering the **lifeline** where the first event occurrence will occur, positioned above that event, in the containing interaction or interaction operand.



5.7 Combined Fragment

Combined fragment is an **interaction fragment**, which defines a combination (expression) of interaction fragments. An interaction operator and corresponding interaction operands define a combined fragment. Through the use of combined fragments the user will be able to describe a number of **traces** in a compact and concise manner.

Interaction **operators** in TDL could be one of:

- **alt - alternatives**
- **opt - option**
- **loop - iteration**
- **break - break**
- **par - parallel**

5.7.1 Alternatives

UML SD:
The interaction operator **alt** means that the combined fragment represents a **choice** or alternatives of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction.

An implicit true guard is implied if the operand has no guard.

An operand guarded by **else** means a guard that is the negation of the disjunction of all other guards. If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing interaction fragment is executed.

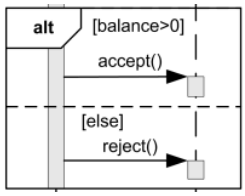


Figure 2: Call accept() if balance > 0, call reject() otherwise.

TDL:
The semantics of the operand without the guard will differ from the UML SD as follows:
If the operand has no guard then the first alternative without guard is executed where the actual message matches the first message of the alternative.

5.7.2 Option

The interaction operator **opt** means that the combined fragment represents a **choice** of behavior where either the (sole) operand happens or nothing happens.

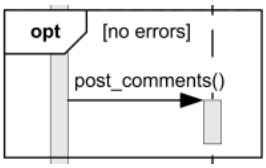


Figure 3: Post comments if there were no errors.

5.7.3 Loop

The interaction operator **loop** means that the combined fragment represents a loop. The loop operand will be repeated a number of times.

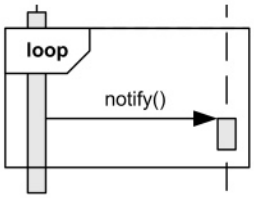
UML SD:
Either or both iteration bounds and a guard could control **loop**.
The loop operand could have iteration bounds, which may include a lower and an upper number of iterations of the loop. Textual syntax of the loop is:

loop-operand ::= **loop** ['(' ***min-int*** [',' ***max-int***] ')']

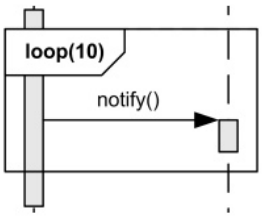
min-int ::= ***non-negative-integer***

max-int ::= ***positive-integer*** | '*'

If loop has no bounds specified, it means potentially infinite loop

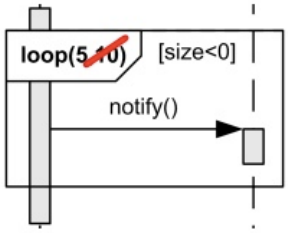


If only ***min-int*** is specified, it means that upper bound is equal to the lower bound, and loop will be executed exactly the specified number of times.



Loop to execute exactly 10 times.

Besides **iteration bounds** loop could also have a **guards**.



Loop is executed 5 times if size < 0
Loop is executed less than 5 times if size becomes >= 0

TDL:
Differences from UML SD are the following:

1) Max-int is not possible

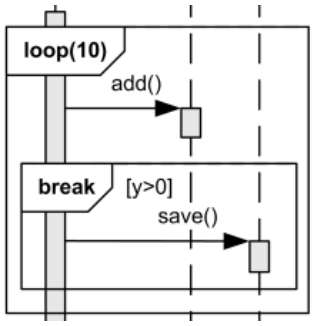
loop-operand ::= **loop** ['(' ***min-int*** ')']

min-int ::= ***non-negative-integer***

5.7.4 Break

The interaction operator **break** represents a **breaking** or exceptional scenario that is performed instead of the remainder of the enclosing interaction fragment.

A break operator with a **guard** is chosen when the guard is true. In this case the rest of the directly enclosing interaction fragment is ignored. When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing interaction fragment proceeds.



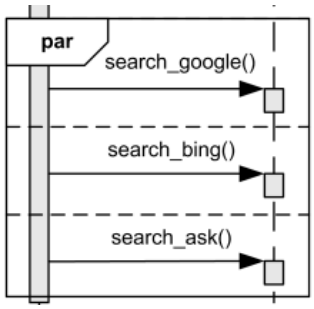
Break enclosing loop if y>0.

A combined fragment with the operator **break** should cover all lifelines of the enclosing interaction fragment.

5.7.5 Parallel

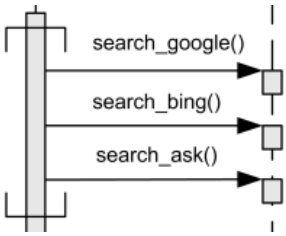
The interaction operator **par** defines potentially parallel execution of behaviors of the operands of the combined fragment. Different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.

Set of traces of the parallel operator describes all the possible ways or combinations that occurrence specifications of the operands may be interleaved without changing the order within each operand.



Search Google, Bing and Ask in any order, possibly parallel.

Parallel combined fragment has notational shorthand for the common situations where the order of events on one **lifeline** is insignificant. In a **coregion** area of a lifeline restricted by horizontal square brackets all directly contained fragments are considered as separate operands of a parallel combined fragment.



Coregion - search Google, Bing and Ask in any order, possibly parallel.

5.8 Message

Message is a **named element** that defines one specific kind of communication between **lifelines** of an interaction. The message specifies not only the kind of communication, but also the sender and the receiver. Sender and receiver are normally two **occurrence specifications** (points at the ends of messages).

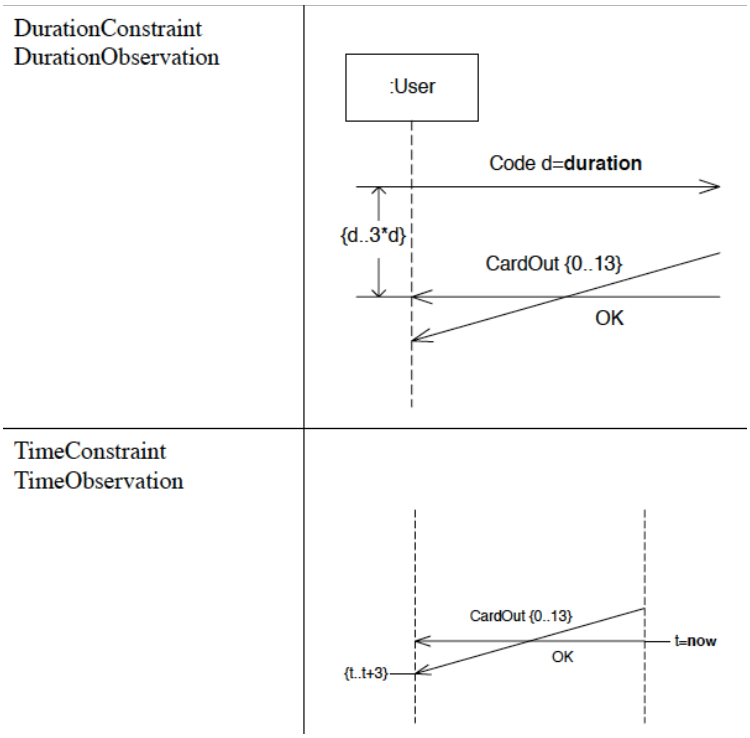
A message is shown as a line from the sender message end to the receiver message end. The line must be such that every line fragment is either horizontal or downward when traversed from send event to receive event. The send and receive events may both be on the same lifeline. The form of the line or arrowhead reflects properties of the message.

A message reflects either an **operation** call and start of execution or a sending and reception of a **signal**.

When a message represents an operation call, the arguments of the message are the arguments of the operation. When a message represents a signal, the arguments of the message are the attributes of the signal.
Depending on the type of **action** that was used to generate the message, message could be one of:

- **asynchronous signal**
- **asynchronous call**
- **synchronous call**
- **reply**

5.9 Time constraints



6 Which UML SD features to leave out of TDL?

1) The following combined fragment operators are irrelevant for TDL:

- strict
- seq
- critical
- ignore
- consider
- assert
- neg

2) Dynamical participants – dynamical creation and destruction of test components.
3) Syntax of the message.
4) Message actions:
    a. create
    b. delete
5) lost and found messages

7 What to take over from TTCN-3 GFT?

7.1 Syntax of the message

Asynchronous signal (message) is defined by

- a data instance/template reference with type information or
- an inline template with type information

Asynchronous and synchronous call are defined by

- keyword “call”
- procedure name
- procedure parameters
    - template reference or
    - list of parameters (can include wildcards, variables and constants)

Reply is defined by

- data instance/template reference or
- inline template

Template matching mechanism (like in TTCN-3) must be used for defining how the actual message field values must be match the expected values.

8 Conclusions

The most important proposals in the document were the following:

1) TDL must be declarative language instead of the algorithmic one.
2) Executable test cases must be possible to generate from TDL automatically.
3) TDL must be independent of executable test cases language.
4) TDL should be based on UML meta-model.
5) It should be easy to use and should include only the most important features of UML SD.
6) For message types and templates references TTCN-3 GFT notation should be used.
7) For validating the test cases the messages order matching and message fields matching mechanisms are used (no explicit verdict clauses).

9 References

[1] OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
[2] http://www.uml-diagrams.org/sequence-diagrams.html