**Data Fuzzing with TTCN-3**

## 1. Scope

Fuzz testing or fuzzing is a well-established automated and efficient black-box testing method for finding software flaws. It is widely used to test for security problems in software or computer systems. It is a black-box testing technique in which the system under test is stressed with invalid, unexpected or random data inputs and data structures through its interfaces. The purpose of fuzzing is to reveal implementation vulnerabilities by triggering failure modes. This is done by stimulating the system with unexpected data in the form of modified valid data, and observing the behavior of the system.

On the other hand TTCN-3 is a widely accepted method in functional (protocol) testing in telecommunications, and pushes into new areas like Intelligent Transport Systems (ITS) or Internet of Things (IoT). TTCN-3 is pretty new to security testing.

Hence we will propose a lightweight extension to the current TTCN-3 standard to support fuzzing with TTCN-3 to maximize its usability for existing TTCN-3 users. Fuzzing operations are defined on basis of the TTCN-3 type system and formally specified by special fuzz functions. The fuzzing itself (i.e. the generation of fuzzed data) is done on the fly during the call of the send operation. The repeated application of data fuzzing, i.e. generation of multiple variants to be sent, can be realized via loop constructs. To allow for deterministic test cases and to support repeatability we are using pseudo randomness specified on basis of a constant seed. While simple dump random fuzzing often causes poor results, intelligent application/protocol based fuzzing is much more powerful. To support application/protocol based fuzz generators fuzz functions can also specified as external functions.

## 2. TTCN-3 Core Language Extensions

### 2.1 Package Conformance and Compatibility

The package presented in the present document is identified by the package tag: "TTCN-3:2013 Fuzz Testing" to be used with modules complying with the present document.

### 2.2 Fuzz Function

A new construct of a fuzz function will be introduced. The concept is similar to the present ordinary functions, but their evaluation is delayed until a specific value is selected via send or valueof operation (lazy evaluation). Fuzz functions may declare formal (in) parameters, and must declare a return type. Apart from the time of evaluation fuzz functions are treated as usual function resp. external functions, hence no extension of the runtime interfaces is required.

Example:

external fuzz function fxz\_UnicodeUtf8ThreeCharMutator(
 in charstring p\_param1) return charstring;

 fuzz function fz\_RandomSelect(
 in integer p\_param1) return integer {
 return rnd(getseed()) \* p\_param1;
}

Generally, matching mechanisms are used to replace values of single template fields or to replace even the entire contents of a template. Matching mechanisms may also be used in-line. A fuzz function instance denotes a set of values, and can only occur in value templates used like a native matching mechanism “instead of values” to define a list of values or templates. It shall not be used on incoming templates.

Example:

template myType mw\_myData := {
 field1 := fxz\_UnicodeUtf8ThreeCharMutator(“abc”),
 field2 := '12AB'O,
 field3 := fz\_RandomSelect((1, 2, 3))

}

A single value will be selected in the event of a sending operation or of the invocation of a valueof operation.

Example:

myPort.send(mw\_myData);

myPort.send(fxz\_UnicodeUtf8ThreeCharMutator(“abc”));

var myType v\_myVar := valueof(mw\_myData);

Storing the selected value of a fuzz function for later use is possible within the sending operation using the ‘-> value’ notation and via explicit invocation of the valueof operation.

Example:

myPort.send(mw\_myData) -> value v\_myVar;

myPort.send(fxz\_UnicodeUtf8ThreeCharMutator(“abc”)) -> value v\_myVar;

var myType v\_myVar := valueof(mw\_myData);

### 2.3 Seed

To allow repeatability of fuzzed test cases, an optional seed shall be used. There will be one seed per test component. Two predefined functions will be introduced to set the seed and to read the current seed value.

setseed([in float initialseed]) return float;

getseed() return float;

Example:

 setseed(1.0);

 var float v\_f := getseed();

Without a previous initialization a value calculated from the system time will be used as seed.