TTCN-3 and Eclipse TITAN for testing protocol stacks

Harald Welte
sysmocom - systems for mobile communications GmbH
Berlin, Germany
laforge@gnumonks.org

Abstract

Implementations of networking protocol stacks are in need of thorough testing in order to ensure not only their security, but also their interoperability and compliance to relevant standards and specifications. Implementing test suites for verification of implementations of TCP/IP communications protocols in the IETF world has traditionally been done in any number of ways, by using general-purpose programming languages such as C, C++, Java, Python and others. In the ITU/ETSI world, TTCN-3 has been develpoed as a domain-specific language for the specific use case of writing protocol conformance tests. This paper acts as an introduction into both TTCN-3 as well as an open source TTCN-3 compiler, Eclipse TITAN.

Keywords

TTCN-3, conformance testing, protocol testing, validation, TITAN

Introduction

Protocol Testing

Testing network protocols is important for a variety of reasons, such as

- conformance to a specification
- ensuring interoperability, including quirks for known-broken other implementations
- network security

The focus of this paper (as well as the focus of TTCN-3) is functional testing. It is not primarily performance testing, though some people use TTCN-3 even in those scenarios. The fact that it's not a scripted language or a VM but a compiled language executing native code helps with that.

In the implementation of a functional test of a given communication protocol, there are typically repeating patterns of certain building blocks, such as

- encoding and decoding messages between their wire format and some higher-level representation
- matching received messages against templates to validate their contents or act depending on various parts of their contents
- waiting for any number of different events/conditions guarded by one or multiple timers

TTCN-3 offers unique language capabilities to address related problems in a very productive and expressive way.

The TTCN-3 Language

TTCN was originally called the *Tree and Tabular combined Notation*, but has since been renamed to the *Test and Testing Control Notation*. It is a purpose-built language with the sole purpose of implementing testing.

History TTCN has been designed as an internationally standardized language purely for testing. Its origins go back to 1883, when ISO was working on Open Systems Interconnection (OSI) conformance testing methodology and framework. It was first standardized as ISO 9646-3 in 1992. Within the ITU / OSI / ISO world, TTCN was well-established and widely used in conformance tests for protocols like those used in ISDN.

In 1997, STF 133 was formed by ETSI MTS to produce TTCN version 3 in co-operation with ITU-T SG10. Main contributions from Nortel, Ericsson, Telelogic, Nokia. More than 200 members participated in the STF133 discussion group, and the language was finally standardized in October 2000.

Ever since, TTCN-3 has gained widespread adoption within the telecom industry. Several major telecom equipment vendors such as Nokia and Ericsson have publicly disclosed some of the extensive in-house testing they do using TTCN-3. A market of TTCN-3 compilers and tools has been well-established, but the proprietary nature of those compilers and toolchains has put TTCN-3 out of reach from the Free Software community.

Meanwhile, as the classic telecom sector became more and more involved with IP and Internet technologies, conformance testing specifications / test suites for Internet centric protocols have been developed by telecom standardization bodies.

For example, ETSI has meanwhile developed and published test suites for IPv6, SIP, DIAMETER, electronic passports, DMR (Digital Mobile Radio) and 6LoWPAN in TTCN-3. Those test suites are available via the ETSI Subversion repository that can be browsed at http://oldforge.etsi.org/websvn/

Also, TTCN-3 test suites have been specified by other bodies for technologies and protocols such as CoAP, MQTT in

the IoT area, or MOST and AUTOSAR in the automotove domain.

Eclipse TITAN

Around 2000, Ericsson internally started development of a TTCN-3 toolchain, which developed into a complete, proven product that was adopted and is used extensively both inside Ericsson, as well as licensed to third parties. Over the years, it has developed into approximately 300,000 lines of Java and 1,6 million lines of C++ code, including extensive self-testing code for TITAN.

TITAN includes not only a TTCN-3 compiler for translating TTCN-3 to C++ and the corresponding runtime libraries, but also a number of other tools, such as a parallel executor for executing test components and result reporting, a Makefile generator, log filtering, report generator, coverage analysis and much more.

Being a commercially developed and supported tool, TITAN also includes a set of extensive user and developer manuals.

In 2015, Ericsson decided to transform TITAN into an open source project under the umbrella of the Eclipse foundation. It is subsequently licensed under Eclipse Public License. Unlike other large software projects developed as proprietary software and dumped to the community, this did not mark the end of Ericsson involvement. To the contrary, the are very active in maintenance of the software ever since, with daily commits to the public git repositories by the Ericsson team, regular releases and participation on forums and mailing lists.

While Eclipse Titan has some optional GUI components within the Eclipse IDE framework, such as the Titan Designer and Titan LogViewer, neither the compiler nor the runtime libraries for executing the actual test suites require those UI components. The entire toolchain can be used from the command line, driven from classic Makefiles.

Together with the TTCN-3 compiler and utilities, Ericsson has also been releasing an ever-growing list of TTCN-3 source code implementations for a variety of protocols. This includes native TTCN-3 implementations of IP, ICMP, IPv6, L2TP, GRE, HTTP, ICMP, RTP, SCTP, SDP, TCP and UDP. It also includes test ports for using the underlaying operating system protocol stack(s), such as the IPL4asp for using the regular socket API as provided by Linux+libc.

Key TTCN-3 Language Features

TTCN-3 is a high-level, abstract language. The code itself is platform independent, as well as test environment independent. In TTCN-3, you define only the abstract messages/signals as they are exchanged between the test system and the tested entity. The transport layers and connections are provided and handled by the tools Message encoding (serialization) and decoding (deserialization) is part of the tool/environment, and not part of the test definition itself.

The strong points of TTCN-3 for use in protocol tests are:

- A rich data/type system
- Parametric templating and powerful template matching
- Behavior specification using the *alt* and *default* behaviors
- Separation of tests, test adapter and codec

TTCN-3 data/type system

TTCN-3 is a strongly typed language. This provides the advantage that many type incompatibilities can already be caught at the compile time, as opposed to weakly typed languages, where type errors are often only discovered at runtime and are hence hard to catch, particularly in rarely used code such as error paths.

Basic types The simple basic types of TTCN-3 include `integer`, `float`, `boolean`.

There `verdicttype` is a special type for storing the preliminary and final verdicts of test execution. It has five distinct values: `none`, `pass`, `inconc`, `fail`, `error`. The useful property is that a verdicttype varialbe can only get *worse*, but never better. So if any part of a test case has ever set the verdict to `fail` or `error`, no follow-up assignments of the variable can ever turn it into `pass` anymore. This greatly simplifies control flow handling in erroneous conditions while writing test cases.

Basic strign types include `bitstring`, `hexstring`, `octetstring`, `charstring` (IA5) as well as `universal charstring` (UCS-4).

Structured Types Using the `record`, `set`, `union`, `record of` and `set of` TTCN-3 structured types, programmers can create abstract container types.

TTCN-3 also knows a `enumberated` type, like many other programming languages.

Not-used and omit Until any variable or field of a structured type has been assigned an explicit value, it is *unbound*. Whenever a value is expected, and that value is unbound, the TTCN-3 runtime will create an error. It's therefore not possible to e.g. accidentially encode unspecified data!

If the programmer wishes to explicitly state that a structured type's optional field is not present, the special value `omit` may be used.

Sub-Typing You can derive a new child type from an existing parent type by restricting the new type's domain to a subset of the parent type's value domain. You can use various sub-typing constructs such as value range, value list, length restriction and patterns, see Listing 1.

Listing 1: Example of TTCN-3 sub-typing

```
type integer MyIntRange (1 .. 100);
type integer MyIntRange8 (0 .. infinity);
type charstring MyCharRange ("k" .. "w");

type charstring SideType ("left", "right");
type integer MyIntListRange (1..5, 7, 9);

type record length(0..10) of integer RecOfInt;

type charstring CrLfTermString (pattern "*\r\n");
```

Parametric templates

When sending messages of a given protocol, templating helps in abstraction, readability and productivity. TTCN-3 has quite extensive templating capabilities.

Templates can be parametric, i.e. they can take arguments just like you would have function arguments in an encoding function in a C program.

Listing 2: Example of parametric templates

```
type record MyMessageType {
  integer field1 optional,
  charstring field2,
  boolean field3
};

template MyMessageType tr_MyTemplate
                (boolean pl_param) := {
  field1 := ?, // present, but any value
  field2 := ("B", "O", "Q"),
  field3 := pl_param
};
```

Templates can be hierarchical, so from the most generic to the most specific case, you can specify ever more concrete templates, depending on the need.

Templates can further be specified as `subset`, `superset`, `permutation` (any order of elements).

Templates can also be used for the receiving side. Here, an incoming, already-decoded message is matched against one or multiple receive templates. Matching against a fixed template wouldn't give much improvement over e.g. comparing with a 'const struct' in a C-language test case. However, TTCN-3 templates can have wild-cards and pattern matching.

The built-in `match()` function can be used to determine if a received message (or actually any value) matches the template. Even more so, the fundamental `receive()` function, which is used to receive any inbound message from a test port, has built-in matching capability, so explicit calls of `match()` are rarely required. See Section 5 below.

TTCN-3 + TITAN encoders/decoders

The data/type system of TTCN-3, extended by non-standard capabilities of TITAN allows describing the messages of practically all possible protocols. The programmer can focus on *expressing* the structure/syntax of the protocol, rather than having to write an encoder/decoder by hand.

TTCN-3 specifies ways of importing other data types or schema definitions, for example ASN.1, IDL, XSD (XML) and JSON. TITAN allows adding encoding instructions as annotations to the TTCN-3 type definitions to automatically encode/decode them into binary or textual forms, XML or JSON or ASN.1 (BER/CER/DER).

Import of the formal definition of a protocol works of course only for such protocols that have a formal definition of their encoding. While this is more often the case in modern telecom protocols, it is not so often the case in the TCP/IP/IETF world. This is where the TITAN binary and text encoder/decoder come into play.

Listing 3 uses the TTCN-3 type language with TITAN binary codec extensions to describe a UDP heaer. We first define a `LIN2_BO_LAST` type as an unsigned 16bit integer with little-endian byte order, then use this to define the UDP header as a record of four such integers to finally define a UDP packet as a record consisting of the header and a variable-length octetstring as payload. Note the `LENGTHTO` and `LENGTHINDEX` notation to express that the "len" field of the header is set to the combined length of the header and the payload of the packet.

Listing 3: Example of TITAN binary codec LENGTHTO

```
type integer LIN2_BO_LAST (0..65535) with
  { variant "FIELDLENGTH(16),
             COMP(nosign),
             BYTEORDER(last)"
  };

type record UDP_header {
  LIN2_BO_LAST srcport,
  LIN2_BO_LAST dstport,
  LIN2_BO_LAST len,
  LIN2_BO_LAST cksum
} with { variant "FIELDORDER(msb)" };

type record UDP_packet {
  UDP_heasder header
  octetstring payload
} with {
  variant (header) "LENGTHTO(header, payload),
                    LENGTHINDEX(len)"
};
```

The next example in Listing 4 contains a partial definition of the GRE header, where individual flag bits at the beginning of the header determine if certain optional fields at some later part in the header are present or not. The `PRESENCE()` attribute of the TITAN binary codec provides an elegant solution to express this:

Listing 4: Example of TITAN Binary codec PRESENCE

```
type record GRE_Header {
  BIT1 csum_present,
  BIT1 rt_present,
  BIT1 key_present,
  ...
  OCT2 protocol_type,
  OCT2 checksum optional,
  OCT2 offset optional,
  OCT4 key otional,
  ...
} with {
  variant (checksum) "PRESENCE(csum_present='1',
                                rt_present='1'B)"
  variant (offset) "PRESENCE(csum_present='1'B,
                             rt_present='1'B)"
  variant (key) "PRESENCE(key_present='1'B)"
```

Much more complex constructs are possible in the TTIAN binary codec, e.g. the extension octet concept found in many telecom protocols.

Abstract Communications Operations

Abstract communications happens on the *test ports* which connect the test case of the abstract test suite (ATS) with the implementation under test (IUT). TTCN-3 supports abstract communications operations for both asynchronous and synchronous communication.

Asynchronous communication is what is typically used to send and receive messages with the IUT. The `send()` function is non-blocking, while `receive()` is blocking[1]. Arriving messages stay in the incoming queue of the destination part. MEssages are sent in order. The receive operation examines the first message of the port's queue, but extracts it *only* if the message matches the receive operations template.

Listing 5: Example receive() operation

```
template MsgType MsgTemplate := { /* valid */ };
var MsgType MsgVar;
PortRef.receive(MsgTemplate) -> value MsgVar;
```

Program Control and Behavior specification

TTCN-3 offers the usual program control statements that C programmers are familiar with: `if`-clauses, `for`-, `while`- and `do-while` loops, including `break` and `continue`. It also offers somehing like the `switch` statement of C, but it is called `select` in TTCN-3. Furthermore, `goto` and associated labels are supported, although again with slightly different syntax than in C.

Beyond those, TTCN-3 offers a couple of unique so-called *behavioral control statements* which are further illustrated below.

The alt statement In testing, quite often one is waiting for one or multiple received messages, guarded by one or multiple timeouts. The blocking semantics of `receive()` means we need some kind of a non-blocking alternative. In TTCN-3, this is achieved by the `alt` statement, which declares a set of alternatives which can happen, but must not happen.

The below example Listing 6 shows a code example that first sends a message through port P, and then waits for either

- a response from port P matching the template `resp`, at which point it will set the verdict of the test to pass and leave the alt.
- any other message on any other port, which it will receive but ignore, and repeat the alt.
- a timeout of the timer T, which will set the verdict to fail and leave the alt.

Listing 6: Example alt statement

```
P.send(req);
T.start;
alt {
  [] P.receive(resp) { setverdict(pass); }
  [] any port.receive { repeat; }
  [] T.timeout { setverdict(fail); }
}
```

The `[]` at the beginning of each line in Listing 6 is the *guard expression* which can be used to restrict whether a given alternative is eligible or not. This can e.g. be used by state machines to allow certain processing only in certain states.

[1] See the `alt` behavioral statement to achieve non-blocking semantics

Experience shows that in more complex tests suites, there will be many alt with partially repeating content, such as e.g. the case when the main execution timer times out, which always leads to fail. Rather than open-coding those alternatives again and again over the code, they can be abstracted out as so-called *altstep*. Those altsteps can then be activated/deactivated and then become active without any explicit code inside each and every `alt`.

The interleave statement In `alt`, one of the stated events must happen in order for the control flow to continue after the statement. The `interleave` statement offers an different behavior in which *all events must happen exactly once, in any order*.

Listing 7: Example interleave statement

```
interleave {
  [] P.receive(1) { ... }
  [] Q.receive(4) { ... }
  [] R.receive(6) { ... }
}
```

Fuzzing extensions

As the TTCN-3 type notation contains stringent information about what values are permitted, it is normally not possible to create illegal values. TITAN introduces the `erroneous` extension which can be usd to generate invalid messages, e.g. those missing mandatory fields, with invalid values, etc.

Test execution

The test suite, comprised of any number of individual test cases is executed by the TITAN executor. The executor takes care of starting any of the parallel test components (on the local host or even on remote hosts), performing the test cases as indicated by either the command line, or by the configuration file, or in absence of that by the compile-time default list of testcases in the (`control`) section of the TTCN-3 code. It also manages opening all configured log files / plugins, as described below. The test executor is typically started using the `ttcn3_start` program.

Configuration File

Every TITAN test suite has a configuration file, where many aspects of test execution can be specified. Among others, you can configure the log verbosity of the TITAN-internal logging. There are different log masks for the log file (`FileMask`) and for the console (`FileMask`). One extremely useful feature is the `TTCN_MATCHING` logging, which will provide detailed information about the exact (even nested) field of a value that did not match in any explicit or implicit matching against a template. Another useful example is `TTCN_ENCDEC`, which will automatically log the input and output of every encode/decode function, which is very useful during debugging of codec problems.

Log Files

TITAN writes log files in a structured format, which makes them easy to parse by downstream log-processing tools. Every parallel test component will write its own log file. As logs

are timestamped, the `ttcn3_logmerge` tool can be used to *splice* the individual log files into each other, providing one log file of all components, sorted by timestamps.

The `ttcn3_logformat` tool can be used to convert the single-line nested value representation into a multi-line format with proper indenting which is more easy to understand as a human. Compare this to e.g. JSON or XML pretty-printing.

Junit-XML output plugin

The TITAN runtime understands the concept of log output plugins. Such plugins can be custom-developed to generate whatever output format from test case logs / results as needed. A number of standard plugins are included, among them the capability to write Junit-XML output, which is a standard format for reporting test results in a machine-readable format. This format is understood by the popular Jenkins continuous integration software, so that one can easily feed per-testcase results into the Jenkins test results analyzer as part of a CI test suite.

Acknowledgments

The author acknowledges that like everyone in Free Software, he is merely standing on the shoulders of giants[2]. Without the work of the ITU and ETSI on the TTCN-3 Language and without the release of Eclipse TITAN as Free Software by Ericsson, he would not have been able to discover TTCN-3.

Special thanks go to Elemer Lelik for his responsiveness and help in getting started with TITAN, as well as the excellent support of the TITAN team for fixing any bugs I reported in virtually no time.

References

[3] TTCN-3 Tutorial ETSI Centre for Testing and Interoperability http://www.ttcn-3.org/files/ETSI_TTCN3_Tutorial.pdf

[2] TTCN-3 Course Presentation Material Ericsson Test Solutions and Competence Center http://www.ttcn-3.org/files/TTCN3_P.pdf

[3] ETSI svn repository http://oldforge.etsi.org/websvn/

[2] In this case, maybe rather *on the shoulders of titans?*