TTCN-3 and Eclipse TITAN for testing protocol stacks

Harald Welte <laforge@gnumonks.org>

Protocol Testing

Important for:

- conformance to specification
- ensuring interoperability
- network security
- regression testing
- performance

Protocol Testing

No standard methodology, language, approach, tool

- testing implementation against itself
  - works only for symmetric protocols
  - wouldn’t cover lots of problems
- testing against wireshark
  - wireshark often way more tolerant than spec
- custom implementation
  - in Python (e.g. using scapy)
  - in Erlang (good binary encoder/decoder) or other languages
- specific tools like packetdrill

Protocol Testing

Personal story: During past years,

- I implemented tons of [telecom] protocols / stacks at Osmocom.org
- I was looking for better tools to help [automatic] testing
  - primarily functional testing (correctness / conformance)
  - not so much performance testing
- I figured Ideal test tool would…
  - allow very productive and expressive way to describe encoding/decoding
  - allow very convenient pattern matching on incoming messages
  - allow exchange of messages asynchronously with implementation under test
- I stumbled on TTCN-3 occasionally and investigated

The TTCN-3 Language

- domain-specific language **just** for protocol conformance tests
- TTCN history back to 1983 (!), TTCN-3 since 2000
- used extensively in classic telecom sector (Ericsson, Nokia, etc.)
- ETSI developed and published abstract test suites in TTCN-3 for
  - IPv6, SIP, DIAMETER, ePassports, Digital Mobiel Radio, 6LoWPAN
- Other bodies published test suites for
  - CoAP, MQTT, MOST, AUTOSAR

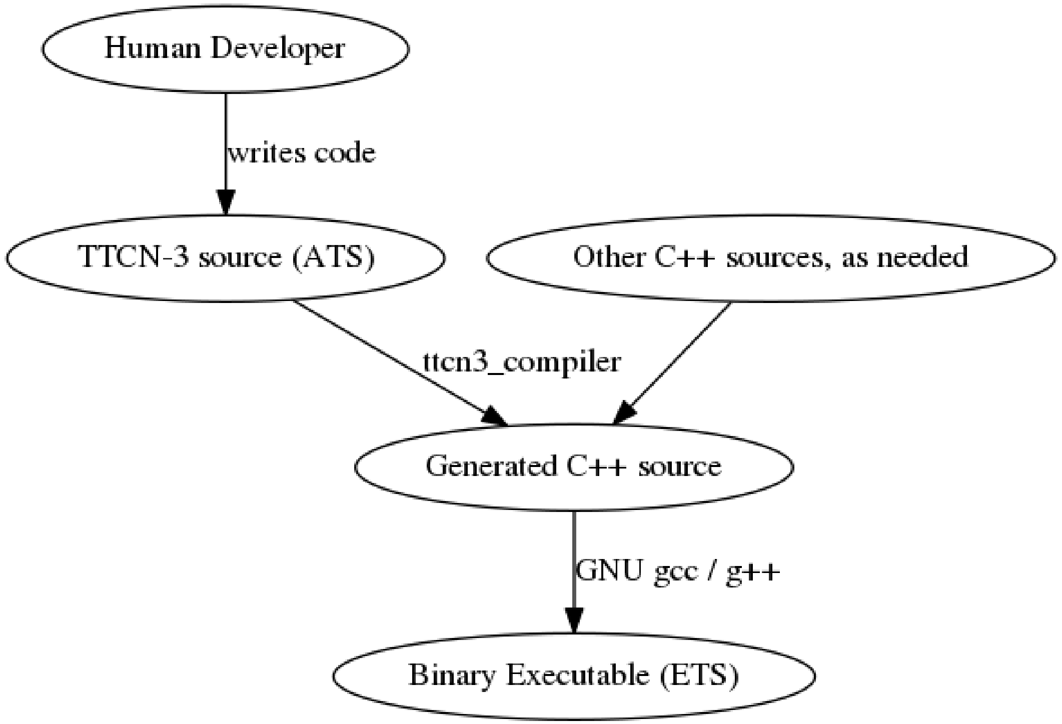
But: Until 2015, only proprietary tools / compilers :(

Eclipse TITAN

- After TTCN-3 specification in 2000, Ericsson internally develops TTCN-3 toolchain
- adopted for many Ericsson-internal testing of all kinds of products
- proprietary software with commercial licenses
- 300,000 lines of Java + 1.6 Million lines of C++
- Released as Open Source as "Eclipse TITAN" in 2015
  - Not just TTCN-3 compiler, but also extensive documentations and many protocol modules, test ports as well as Eclipse IDE, Log file viewer/visualizer, etc.
- `eclipse-titan` part of standard Debian / Ubuntu archive, only one apt-get away

Great, we can finally use TTCN-3 in FOSS!

Eclipse TITAN compiler workflow



- TITAN actually *compiles* into executable binaries, it is not using a VM or scripting
  - ATS: Abstract Test Suite (source code)
  - ETS: Executable Test Suite (executable code)

TTCN-3 Language Features (with TITAN)

- comprehensive type system
- parametric templates
- variety of encoders/decoders
- automatic / comprehensive logging framework
- powerful program control statements
- built-in notion of tests cases, test suites, verdicts, ...
- runtime / executor for parallel test components + aggregating results

TTCN-3 Basic Types

- Simple basic types such as `integer`, `float`, `boolen`
- Basic string types such as `bitstring`, `octetstring`, `hexstring`, `charstring` (IA5) and `universal charstring` (UCS-4).
- Structured Types `record`, `set`, `record of`, `set of`
- Verdict type `verdicttype`
  - can have either value `none`, `pass`, `inconc`, `fail`, or `error`
  - verdict can only *deteriorate* (`pass` → `fail`) but never improve (`error` → `pass`)
  - every test case implicitly has a verdict, no need to explicitly declare a variable of `verdicttype`

TTCN-3 Structured Types

A structured type is an abstract type comprised of other types, whcih can be nested. An example for a `record` type (similar to a C-language `struct`) is shown below

```
type record MyMessageType {
  integer field1 optional<1>,
  charstring field2,
  boolean field3
};
```

1. optional members may be present or not

TTCN-3 Union Type

A union expresses a set of alternative types of which one alternative must be chosen.

```
type union MyMessageUnion {
  integer field1,
  charstring field2,
};
```

Difference to C-language union: `ischosen()` can be used to learn which of the union members is chosen/defined!

Not-used and omit

- until a variable or field of structured type is assigned, it is *unbound*
- whenever a *value* is expected, TTCN-3 runtime will create an error for *unbound*
- in case of absence of optional fields, explicit `omit` value must be assigned!

Sub-typing

Sub-typing can be used to further constrain a given type. Typical examples include constrained number ranges, and string patterns

```
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 CrLfTermStrin (pattern ”*\r\n”);
```

Templates

- Matching incoming messages against some kind of specification is one of the most common tasks in testing protocols
  - some expected fields are static (message type)
  - some expected fields are known (source address)
  - some fields are chosen by sender (some identifier)
  - some fields we don’t care (optional headers that may or may not be present)
- TTCN-3 Templates provide elegant solution for this, avoiding any explicit code to be written
  - templates can even be parametric, i.e. they can be instantiated with "arguments"
- templates can also be used for sending messages, if they are fully specified/qualified

Templates

```
// Value list template
template charstring tr_SingleABorC := ("A", "B", "C");
```

```
// Value range
template float tr_NearPi := (3.14 .. 3.15);
template integer tr_FitsToOneByte := (0 .. 255);
template integer tr_GreaterThanZero := (1 .. infinity);
```

```
// Intermixed value list and range matching
template integer tr_Intermixed := ((0..127), 200, 255);
```

Matching inside values

```
// Using any element matching inside a bitstring value
// Last 2 bits can be '0' or '1'
template bitstring tr_AnyBSValue := ’101101??’B;
```

```
// Matches charstrings with the first character "a"
// and the last one "z"
template charstring tr_0 := pattern "a*z";
```

- more capabilities using `complement`, `ifpresent`, `subset`, `superset`, `permutation` constructs not covered here

Parametric Templates

See below for an example of a parametric template:

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

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

The built-in `match()` function can be used to check if a given value matches a given template. Some TTCN-3 statements such as `receive()` have built-in capabilities for template matching, avoiding even the explicit call of `match()` in many cases.

Template Hierarchy

Using modified templates, one can build a hierarchy of templates: From the specific to the unspecific

```
template MyMsgType t_MyMsgAny := {
  msg_type := ?,
  foo := bar
};

template MyMsgType t_MyMsg23 modifies t_MyMsgAny := {
  msg_type := 23,
};
```

where

- *t_MyMsgAny* matches a message with any message type and "foo=bar", while
- *t_MMyMsg23* matches only those that have "foo=bar" and "msg_type=23"

Encoders/Decoders

- type system, templates, matching are all nice and great, but we need to get data from wire format into TTCN-3 abstract types
- TTTCN-3 specifies importing of formal schema definitios, such as ASN.1, IDL, XSD (XML) and JSON
- TITAN has additional codecs for those (many) protocols that lack formal syntax
  - `raw` codec for binary protocols (e.g. GTP)
  - `text` codec for text based protocols (e.g. HTTP, MGCP, IMAP, ...)
- codecs allow you to *express/describe* the format (declarative programming) rather than the usual imperative approach

TITAN raw codec: UDP Example

How to express an UDP header using TITAN raw codec

```
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_header header
  octetstring payload
} with {
  variant (header) ”LENGTHTO(header, payload), LENGTHINDEX(len)”
};
```

TITAN raw codec: GTP Example

How to express an GTP header using TITAN raw codec

```
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)"
}
```

TITAN text codec: MGCP Example

```
type charstring MgcpVerb ("EPCF", "CRCX", "MDCX", "DLCX", "RQNT",
"NTFY",
                          "AUEP", "AUCX", "RSIP") with {
  variant "TEXT_CODING(,convert=upper_case,,case_insensitive)"
};
type charstring MgcpTransId     (pattern "\d#(1,9)");
type charstring MgcpEndpoint    (pattern "*@*");
type charstring MgcpVersion     (pattern "\d.\d") with {
  variant "BEGIN('MGCP ')"
};
type record MgcpCommandLine {
        MgcpVerb        verb,
        MgcpTransId     trans_id,
        MgcpEndpoint    ep,
        MgcpVersion     ver
} with {
        variant "SEPARATOR(' ', '[\t ]+')"
        variant "END('\r\n', '([\r\n])|(\r\n)')"
};
```

Program Control Statements

- `if` / `else` like in C
- `select` statement similar to C `switch`
- `for`, `while`, `do-while` loops like in C
- `goto` and `label`
- `break` and `continue` like in C

Abstract Communications Operations

- TTCN-3 test suites communicate with *implementation under test* through abstract TestPorts
  - TestPorts can be implemented in TTCN-3 or C++ and linked in
  - TestPorts must be *connected* before using send/receive operaitons
  - TITAN provides TestPorts for e.g. packet socket, IP/UDP/TCP/SCTP socket, ...
- `<port>.send(<ValueRef>)` performs non-blocking send
  - Literal value, constant, variable, specific value template, ...
- `<port>.receive(<TemplateRef>)` or `<port>.receive` performs blocking receive
  - literal value, constant, variable, template (with matching!), inline template

'... but if receive blocks, how can we wait for any of N events?

Program Control and Behavior

- program statements are executed in order
- blocking statements block the execution of the component
- occurrence of unexpected event may cause infinite blocking

```
// x must be the first on queue P, y the second
P.receive(x); // Blocks until x appears on top of queue P
P.receive(y); // Blocks until y appears on top of queue P
// When y arrives first then P.receive(x) blocks -> error
```

This is what leads to the `alt` statement: `alt` declares a seto alternatives covering all events, which

- can happen: expected messages, timeouts, ...
- must not happen: unexpected faulty messages, no message received, ...
- all alternatives inside `alt` are blocking operations

The `alt` statement

```
P.send(req)
T.start;
// ...
alt {
[] P.receive(resp)  { /* actions to do and exit alt */ }
[] any port.receive { /* handle unexpected event */ }
[] T.timeout        { /* handle timer expiry and exit */ }
}
```

- [] is guard condition enables or disables the alternative
    - usually empty `[]` equals `[true]`
    - can contain a condition like `[x > 0]`
    - very good for e.g. state machines to activate some alternatives only in certain states while others may occur in any state

The `alt` and `repeat` statements

The `repeat` statement

- takes a new snapshot and re-evaluates the alt statement
- can appear as last statement in statement blocks of statements

```
P.send(req)
T.start;
alt {
  [] P.receive(resp)        { /* actions to do and exit alt */ }
  [] P.receive(keep_alive) { /* handle keep alive message */
                              repeat }
  [] any port.receive       { /* handle unexpected event */ }
  [] T.timeout              { /* handle timer expiry and exit */ }
}
```

TTCN-3 modules

TTCN-3 code is written in *modules*

- a test suite consists of one or more modules
- a module contains *module definitions* and an optional *control part*
  - *parameters* (automatically configurable via config file)
  - definition of *data types*, *constants*, *templates*
  - definition of *communications ports*
  - definition of *test components*, *functions altstesp* and *test cases*
  - *control part* determines default order/execution of test cases
- modules can import from each other (think in python terms)

Examples

Let’s have a look at some real-world examples and do a bit of a walk-through before continuing with the slides…

Logging

- TITAN runtime contains extensive logging framework
- config file determines log level for various different subsystems
  - e.g. any encode, decode, receive, transmit operations logged
  - timer starts, expirations
  - any changes to test case verdict
- explicit logging from code by use of `log()` built-in function
- `ttcn3_logformat` tool for pretty-printing log files
- `ttcn3_logmerge` tool for merging/splicing multiple logs
- log plugins e.g. for generating JUnit-XML available
  - facilitates easy reporting / integration to Jenkins or other CI

Logging

Log file format example:

```
// abstract data type before encode
13:30:41.243536 Sent on GTPC to system
@GTP_CodecPort.Gtp1cUnitdata : { peer := { connId := 1, remName :=
"127.0.23.1", remPort := 2123 }, gtpc := { pn_bit := '0'B, s_bit
:= '1'B, e_bit := '0'B, spare := '0'B, pt := '1'B, version :=
'001'B, messageType := '01'O, lengthf := 0, teid := '00000000'O,
opt_part := { sequenceNumber := '3AAC'O, npduNumber := '00'O,
nextExtHeader := '00'O, gTPC_extensionHeader_List := omit },
gtpc_pdu := { echoRequest := { private_extension_gtpc := omit } }
} }

// 'msg' contains encoded binary data actually sent via socket
13:30:41.243799 Outgoing message was mapped to
@IPL4asp_Types.ASP_SendTo : { connId := 1, remName :=
"127.0.23.1", remPort := 2123, proto := { udp := { } }, msg :=
'32010004000000003AAC0000'O }
```

Logging

The same log file lines if run through `ttcn3_logformat`

```
13:30:41.243536 Sent on GTPC to system
@GTP_CodecPort.Gtp1cUnitdata : {
    peer := {
        connId := 1,
        remName := "127.0.23.1",
        remPort := 2123
    },
    gtpc := {
        pn_bit := '0'B,
        s_bit := '1'B,
        e_bit := '0'B,
        spare := '0'B,
        pt := '1'B,
        version := '001'B,
        messageType := '01'O,
        lengthf := 0,
        teid := '00000000'O,
        opt_part := {
            sequenceNumber := '3AAC'O,
            npduNumber := '00'O,
```

```
            nextExtHeader := '00'O,
            gTPC_extensionHeader_List := omit
        },
        gtpc_pdu := {
            echoRequest := {
                private_extension_gtpc := omit
            }
        }
    }
}
13:30:41.243799 Outgoing message was mapped to
@IPL4asp_Types.ASP_SendTo : {
    connId := 1,
    remName := "127.0.23.1",
    remPort := 2123,
    proto := {
        udp := { }
    },
    msg := '32010004000000003AAC0000'O
}
```

Existing TITAN Source

- Protocol encoding/decoding
  - BSSAP+, BSSGP, BSSMAP, CoAP, DSS1, DUA, EAP, GRE, GTP, HTTP, ISUP, LLC, M2PA, M2UA, MQTT, MongoDB, NDP, NS, NTAF, ROSE, SCTP, SDP, SNDCP, STOMP, STUN, SUA, TLS, WTP, DNS, IP, SMPP, SNMP, IKEv2, DHCP, PPP, RTP, TCP, UDP, XMPP, DHCPv6, SMTP, ICMP, RTSP, ICMPv6, DIAMETER, FrameRelay, ProtoBuff, IUA, L2TP, M3UA, MIME, WebSocket, H.248, IMAP, IPsec, SRTP, MSRP, ICAP, RADIUS
- Protocol Emulation
  - M3UA, SCCP, SUA
- Test Ports
  - GPIO, MTP3, Serial, SocketCAN, SCTP, SIP, HTTP, Telnet, UDP, pcap file, pipe, SQL, TCP, SUNRPC, SSH, STDINOUT, sockets, LDAP

Further Reading

- Ericsson TTCN-3 tutorial http://www.ttcn-3.org/files/TTCN3_P.pdf
- An Introduction to TTCN-3, 2nd Edition http://www.wiley.com/go/willcock_TTCN-3_2e
- Modules https://github.com/eclipse
- More Modules http://git.eclipse.org/
- Debian https://packages.debian.org/search?keywords=eclipse-titan
- Ubuntu https://packages.ubuntu.com/search?keywords=eclipse-titan

EOF

End of File