Testerman

- A multi-purpose tool for multi-domain testing
  - Inspired by TTCN-3 standard
    - Most TTCN-3 primitives implemented as Python modules (libs)
  - Initially designed to test VoIP platforms, natively extendable to most other technical environments
  - Enables automated integration tests such as
    - Provisioning through an API (WS, XML over HTTP, ...)
    - Actual calls (SIP, Sigtran, ...)
    - CDR andlog checking
  - Can be used for network interfaces unit testing
    - In particular in WS APIs (using CRUD approaches)
- A tool to create reproducers
  - Special network messages, etc
- A framework to develop simulators
  - Stubs & drivers
  - Network elements
  - Softphones, Diameter servers, HLR, …

Testerman, What for ?

- Suitable for
  - Integration tests (multiple “domain” testing: VoIP, SOAP, API, AAA, …)
  - Unitary tests using network connections
  - Campaign-based testing for Regression
  - Application prototyping / simulators
- Not suitable for
  - Load tests (not yet)
  - Binary (non-network) API testing, though TTCN-3 provides a way to support it
    - As a consequence, won't replace basic xUnits
  - Strict protocol/conformance testing

Why Another Testing Tool ?

- Existing testing products only cover a particular test domain
  - Telecommunication protocols, or Web Services, or Web, or …
  - But never Telecom + Web Services + SNMP + SSH (to simulate some user actions)

- Testerman is basically a framework that work with neutral scripts
  - A kind of “pivot format” (whose syntax is independent from the testing protocols) is used
  - TTCN-3 (Testing & Test Control Notation) is a language to describe such tests
  - We use the same TTCN-3 concepts, with a Python syntax
    - No TTCN-3 compiler required
    - We dropped the heaviness of TTCN-3 strong typing
    - (That's why conformance protocol testing is not as adapted as TTCN-3)

Why Choosing Testerman ?

- Based on the same concepts as TTCN-3
  - Proven model, not reinvented
  - Shared skills
    - Once you learned Testerman, you can easily adapt to TTCN-3
    - If you already know TTCN-3, you'll adapt to Testerman as easily

- Lighter than TTCN-3
  - Tests are created in less time
  - More adapted to non-strict protocols

- Easily extensible
  - Language-independent
    - You may create new probes in any language
      - Stubs available in Python for now, C++ and Java on the way
  - Open-source

- Ready-to-use implementation, several test adapters and codecs already running
  - Probes: tcp, udp, sctp, diameter, rtp, soap, xmlrpc, ssh, sql
  - Codecs: xml, http, [sip]

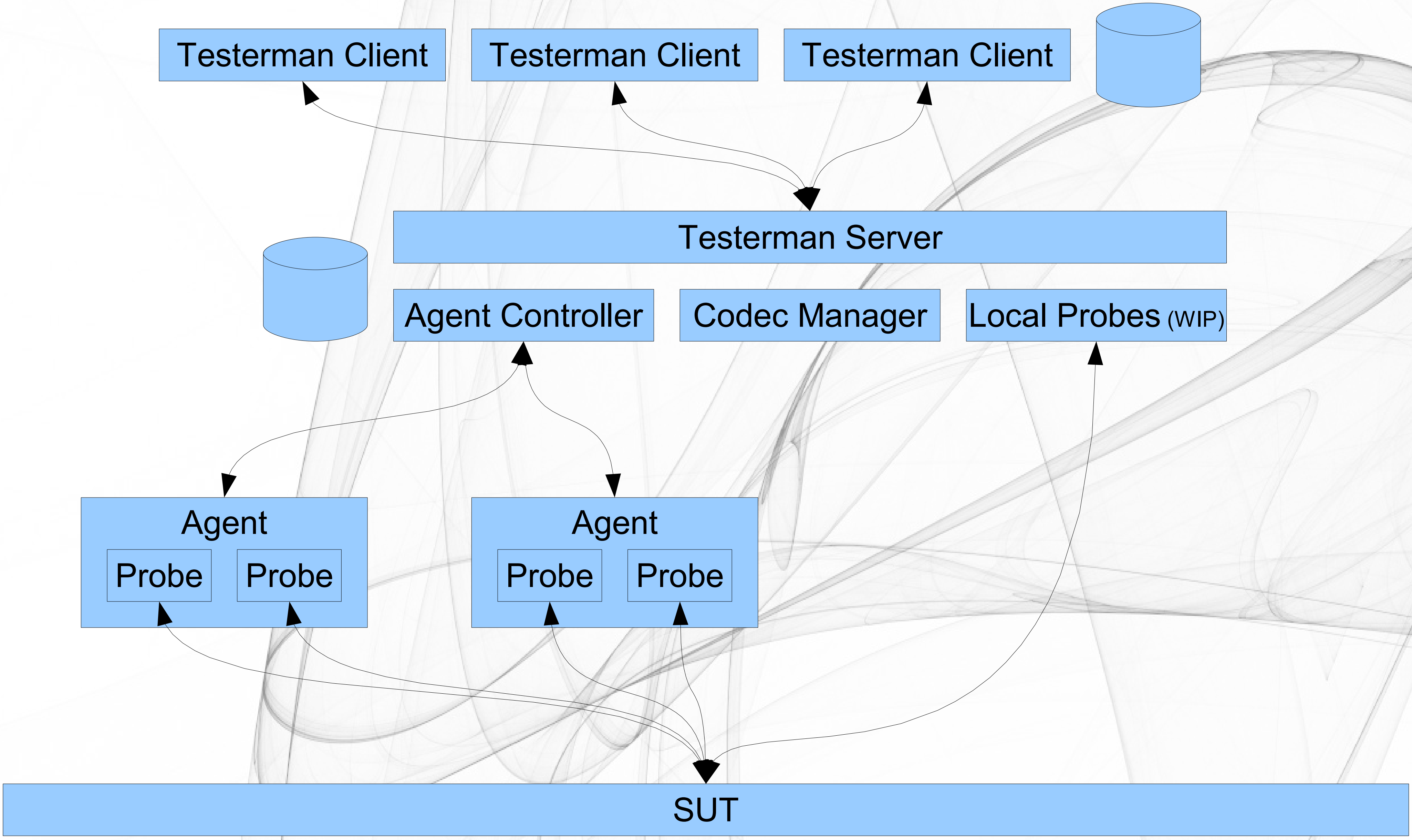
- Can be used as a platform to develop simulators and application prototypes

Testing Concepts

- Test process:
  - Something is tested (the “system under test” - SUT)
  - By something else (the “tester”)
  - Using interactions (the “connections”)

  - We send stimuli towards the SUT
  - We expect some reactions from the SUT
  - If we get actual reactions
    - They may match the expected ones (including timeouts): test passed
    - They don’t match the expected ones: test failed
- Testerman enables to
  - Stimulate any SUT interfaces
  - Expect reactions on any SUT interfaces
    - Providing a probe implementation exists for this interface
    - (Probes are TestAdapter in TTCN-3)

Testerman Architecture



Testerman Clients

- Currently only a CLI client is available
  - Ideal to embed it into Makefiles, continuous integration systems, ...
- A PyQt-based client is under dev
  - Multi-platform
  - IDE
  - ATS/Campaign/Packages browsing
  - Log analyser (and exporters)
  - Job control
- Everything is executed by the server
  - No dependency on client's network connectivity
  - Useful for long campaign (several hours)

Testerman vs TTCN-3

- TTCN-3
  - Strong message typing (ASN.1 like)
    - For all internal or external messages
  - Is a pure abstract model, not an implementation
  - Requires a TTCN-3 compiler
    - That turns the ATS into a machine-compilable code (Java, C++, …)
    - Then we need to bind/link it with TestAdapters (TA) implementations
    - And recompile the whole thing to create a TestExecutable (TE)
- Testerman
  - No strong typing, messages are directly valued with their structure
    - The typing may be checked by the probes/codecs
  - Is a complete implementation with a Distributed TestAdapter implementation
    - The TA is the set of available probes, codecs, and the remote probe controller
  - The ATS is an unmodified part of the TE
    - Testerman just adds some stubs around it to create the complete TE
  - Does not support all TTCN-3 primitives, however

TTCN-3 Sample (echo)

```
type record EchoRequest
{
        charstring data,
}

type record EchoResponse
{
        charstring data
}

template EchoRequest echo_request(something)
{
        data := something,
}

template EchoResponse echo_response
{
        data := "hello world"
}


// We first define a Port type, telling what messages can be sent/received
// through this port type.

type port SelfcarePort message {
        out EchoRequest;
        in EchoResponse;
}

// then we declare a component type, interfacing this port.
type component EchoComponent {
        port EchoPort echo;
}

// System type: mandatory in TTCN-3, unused in Testerman
type component SystemType {
        port EchoPort system system_echoPort;
}
…
```

```
…
testcase echo() runs on EchoComponent system SystemType
{
        log("Attempting to open a selfcare session...");

        map(mtc:echoPort, system:system_echoPort);

        echoPort.send(echo_request("hello world"));

        alt
        {
                [] echoPort.receive(echo_response)
                {
                        log("OK");
                        setverdict(pass);
                }
                [] echoPort.receive
                {
                        setverdict(fail);
                }
        };

        stop;
}

control {
        execute {
                echo()
        }
}
```

+ external system definition (test adapters mapping to system ports)

Testerman Sample (echo)

```
def echo_request(something):
        return something

def echo_response():
        return "hello world"

class TestCase_Echo(TestCase):
        def body(self, vars):

                log("Echoing...")

                p = self.mtc['echoPort']
                port_map(p, self.system['echo'])
                p.send('hello world')


                log("waiting for a response...")

                alt([
                 [ p.RECEIVE(echo_response()),
                   lambda: self.log("OK"),
                   lambda: self.setverdict("pass"),
                 ],
                 [ p.RECEIVE(),
                   lambda: self.setverdict("fail")
                 ],
                ])

                log("SUCCESS")


# Addon: system setup
system = System()
system.requires('echo', 'local.echo/echo')

# Control part
TestCase_Echo().execute(system = system)
```

Writing Campaigns

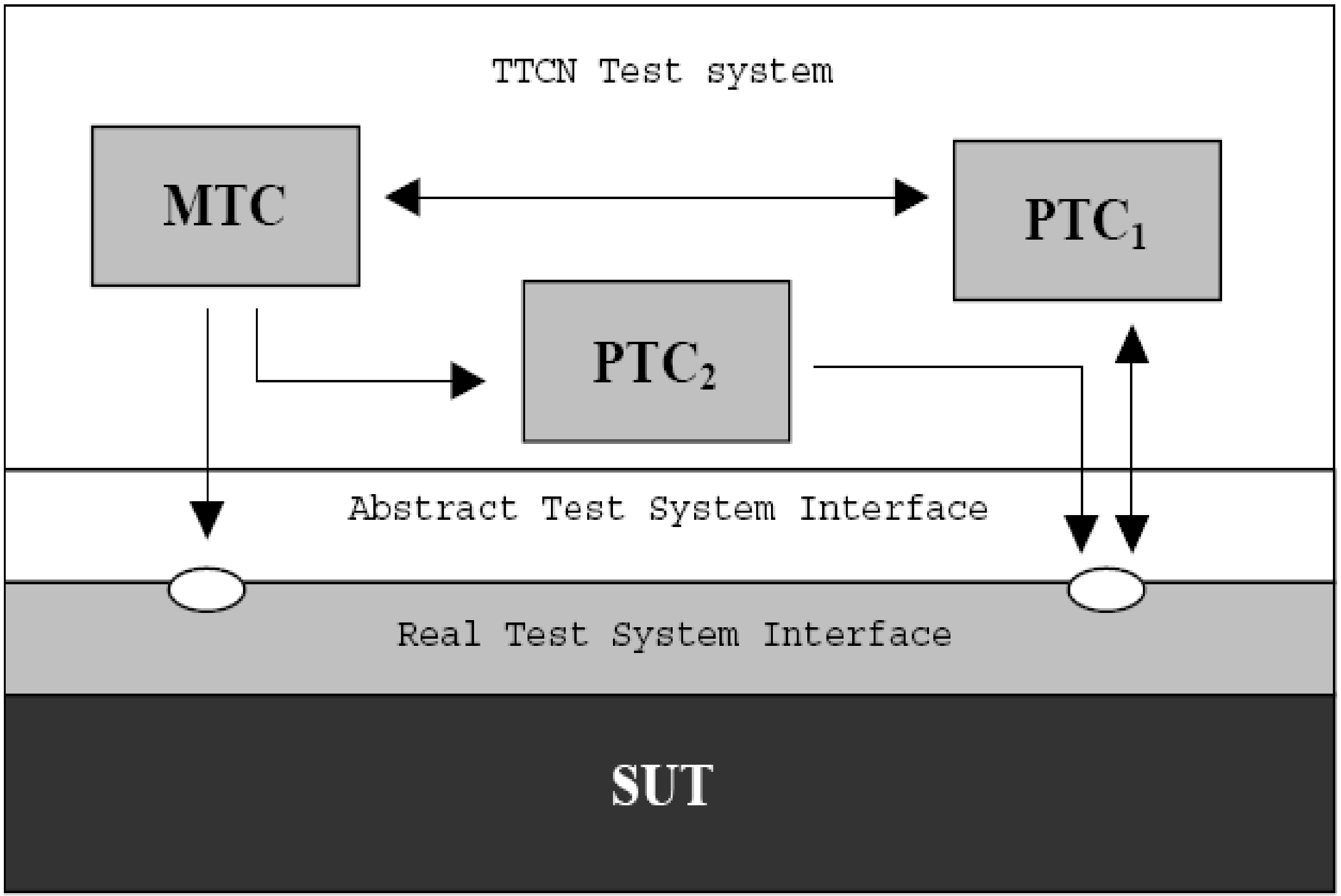
- TDB

TTCN-3 Concepts

- Testerman implements the following TTCN-3 Concepts
  - Test Components (TC), with Ports
  - Behaviour
  - Timers
  - TTCN-3 messaging system based on messages
    - Only asynchronous messages, no « API call »
    - Sending/receiving messages
    - Template matching
    - Template value
  - Alt statement
  - TestCase
  - Verdict management

  - And everything basically provided by Python (functions, altsteps, conditions, loops...)

Test Components

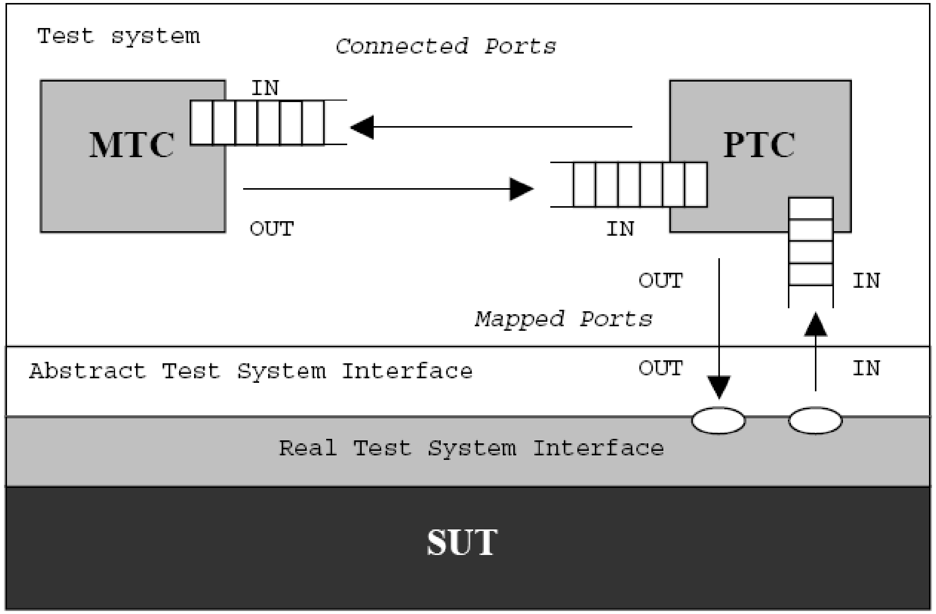


Test Components (2)

- What is a TC ?
  - An abstract entity that has its own live within the TTCN-3 world
  - A TC exposes ports that can be connected together (on itself or on other TC)

  - MTC
    - Main Test Component
      - Automatically created, runs the body() of a TestCase
  - PTC
    - Parallel Test Component
      - Created on demand, used to run what TTCN-3 calls a Behavior
        - (in its own pseudo-thread – may be distributed on any Testerman TE Node)
    - Manages a local verdict
    - Terminated at the end of the testcase
      - The final TestCase verdict is the combination of all local verdicts (PTC and MTC)
    - May be created with a « alive » flag or not (default)
      - Alive practically means: « can be reused to run other behaviors »
  - System
    - A special Test Component that represents the Test System Interface (TSI)
    - Automatically created if not provided, available within a test case context

Connections



Connections (2)

- Two kinds of connections
  - Connection between 2 TC
    - Both way connections
    - 1-to-1
    - 1 to many
    - Loopback
    - TTCN-3 constraints, see next slides
    - connect/disconnect operations in Testerman
- Connection between a TC and TC:System
    - The operation is then called a « mapping »
    - Associate a TC port to Test System Interface (TSI) port
      - i.e. to a probe instance
    - port_map/port_unmap operations in Testerman

Component Connections

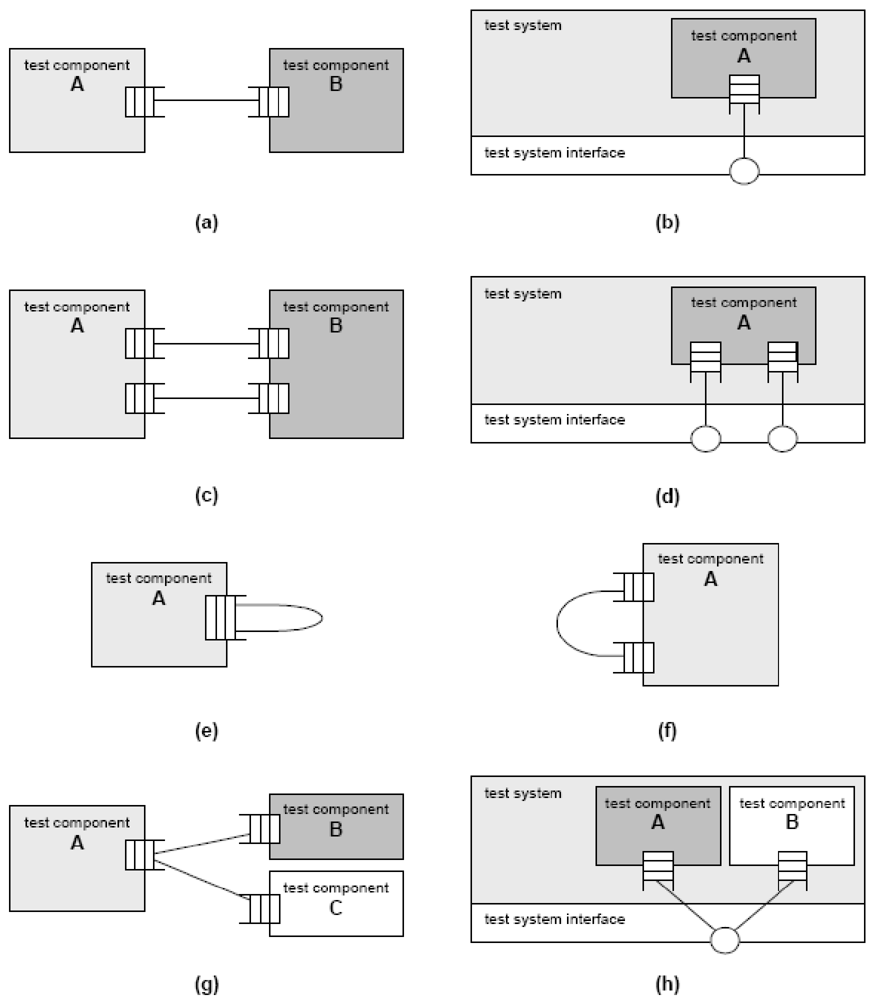


Figure 5: Allowed connections

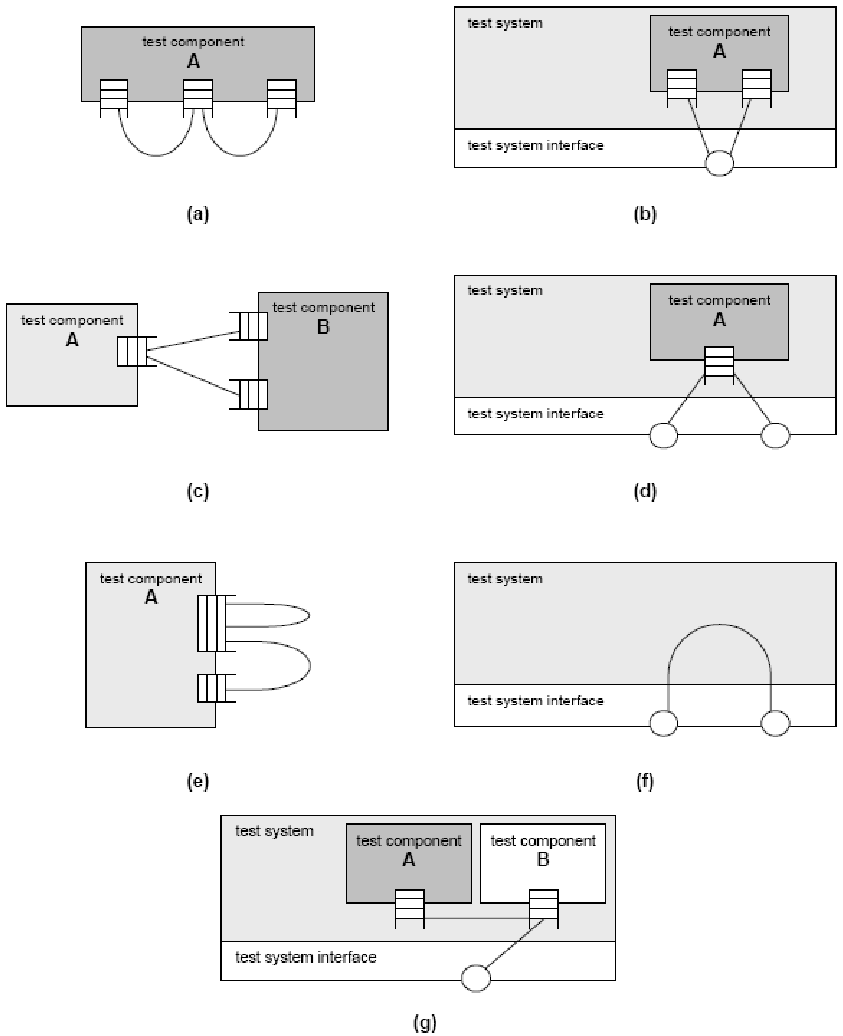


Figure 6: NOT allowed connections

Mapping

- Additional abstraction level
  - The actual system / probe connections are prepared out of a testcase definition
  - mySystem.requiresProbe('sip01@remote.voipsip', 'sip')
  - TestCase(system = mySystem)
    - port = mtc['registration']
    - map(port, mySystem['sip'])
- The probe availability is checked on requires()
  - Probe lock management

Benefits of this Model

- The test logic is built independently from the probes
  - Your test state machines are not mapped to the probes ones
  - High flexibility
    - Multiple TCs may watch system ports
  - Communications between components
    - Enables synchronization between a RTP state machine and a SIP state machine, for instance
    - Messages exchanges at this level are free
      - Testerman implementation won't check types
      - In a real TTCN-3 implementation, the test engineer is responsible for defining what he/she needs, so it's free, too
- Parallel state machines
  - Running a Behavior into a PTC
  - Simplifies lots of ATSes
    - For example, « A calls B »
    - We program A's caller state machine (full)
    - We program B's called state machine (full)
    - Run A's behavior within PTC01, B's within PTC02
    - Just wait for the final verdict (from all PTCs)
- Naturally embeds « simulators » into test designs

Test Component Management

- Test Components are only created within a TestCase.body()
  - self.system and self.mtc are automagically created
  - To create a (P)TC: self.create() or self.create(name = None, alive = False)
    - By default, a name is autogenerated (component_N)
    - alive is set to False
- With these new objects, the following methods are available:
  - start(Behavior(), args = None)
  - stop()
  - kill() # you shouldn't use this
  - alive(), running()
  - done()
  - Events (to use in alt): .KILLED, .DONE
  - port['something'] or ['something']
    - Returns a Port object; the port is a full TTCN-3 abstraction and not attached to any probe implementation
    - Since we don't define, in Testerman, the available ports for a component, you can get as many ports as you need (or want)

Timers

- Dedicated object
  - t = Timer(10.0) or t =Timer()
  - t.start() or t.start(10.0)
  - t.timeout()
  - t.stop()
    - Useless if expired, safe even if stopped
  - t.TIMEOUT event in alt()
  - All (running) timers are automatically stopped at the end of a TestCase

Templates

- Templates are the way messages are described
  - Used for messages to send
    - Fully qualified templates
      - No ambiguity: all required message fields are filled
  - And for message to receive (template matching)
    - Use conditions to match a message

- Message structures
  - In TTCN-3, messages are strongly typed
  - In Testermain, no typing
    - You directly provide the structure you want, with its values

Templates: Structures

- Messages can be simple
  - Integer, float, string
- Or structured
  - If you have in mind the ASN.1 typing of your struct, just translate it into Testerman:
    - ASN.1 SEQUENCE -> Python dictionary [C struct]
    - ASN.1 SEQUENCE OF -> Python list
    - ASN.1 CHOICE -> Python couple
  - Recursive, of course (list of dict containing couples containing simple, leaf types)

Templates: Samples

Simple types

```
# String
MyMessage = "This is a simple string"
# Unicode string
MyMessage = u"Une chaîne française"
# Numerics
MyMessage = 120
MyMessage = 120.0
```

Structured types

```
# a struct
MyMessage = { 'field1': 'value1', 'field2': 120 }
# a list (please use the same element type)
MyMessage = [ 'element1', 'element2', 'element3']
# a choice: we selected the 'choiceName' name
MyMessage = ('choiceName', 'value')
MyMessage = ('choiceName', 123)
```

More structured types

```
MyMessage = ( 'ClassName', { 'member1': 'value1', 'member2': 120 } )
MyMessage = { 'line_ids': [ 1, 1334, 23231 ] }
MyMessage = [ ('ArgClass1', { } ), ('ArgClass2', 'subarg': ('SubClass1', { 'tok': '12ab43' } )) ]
```

Template Matching

- Fully qualified templates are complete messages
  - This kind of template can be send through a TC-port
- When receiving a message from a TC-port
  - You must be able to check if it is what you were expecting
  - Template matching
  - Received messages are fully qualified
    - And are checked against your templates
- Comparators/matchers are available
  - greaterThan, lowerThan, contains, regexp, notPresent, …

```
MyExpectedMessage = greaterThan(5) # match a numeric >= 5


# requires m2 to be present, any value, and m2 not to be provided.
MyExpectedMessage = { 'm1': 'value1', 'm2': any(), 'm3': notPresent() }
```

Template Matching: Comparators

- Todo

Templates Implementation

- Best practices:
  - All reusable templates should be written as a function
    - Makes it parameterized
    - And available in all scopes
  - You may use local functions for non-shared templates

```
def connectRequest(login, pwd):
  return { 'operation': 'connect', 'login': login, 'password': pwd }

def connectResponseOk():
  return { 'status': 200, 'type': 'connectResponse' }
```

Value Extraction

- When matching an incoming message, you may need to retrieve some of its values
  - SessionId from a http response,
  - Fields from a SIP message, …
- TTCN-3 proposes the following mechanism:

```
port.receive(template) -> value m ;
log("received:" & m);
```

  - Translated into Testerman:

```
port.receive(template, 'm')
log("received:" + value('m'))
```

  - value('m') is the complete received message, you may traverse it

```
value('m')['field1'] …
```

- Usable in alt statement, too

Alt Statement

- A Kind of asynchronous switch/case
  - Blocking call
  - Waiting for an event (matched template)
  - That may occur on any watched ports (any TC) or timers
  - Once an event is matched, a sequence of instructions is executed (the altstep)
  - Then returns, or may be repeated

  - If multiple events are received on a single port, the first matched event is handled first (no best matching against templates)
  - Events arrived on watched ports and not matched are deleted
  - Events arrived on non-watched ports are enqueued in background

- Warning
  - If no matched event occurs, only interruptible with a tc.kill() operation
  - Use watchdog timers in most of your operations

Writing Alt statement

Alt statement:

- alt(list of clauses)
- clause = [guard] <event to match> [action list]

Usable in:

- Testcase.body
- Behavior.body

Testerman syntax: (WARNING: subject to change for now)

```
alt([
[ port.RECEIVE(mymessage),
  lambda: self.setverdict("pass"),
],
[ lambda: a > 1,
  port.RECEIVE(),
  REPEAT,
],
[ port2.RECEIVE(anothermessage, 'keepme'),
  lambda: log("Sorry, received invalid message" + str(value('keepme'))),
  lambda: self.setverdict("inconc"),
],
[ timer.TIMEOUT,
  lambda: self.setverdict("fail"),
  lambda: stop(),
]
])
```

Writing Alt statement (2)

- Guard (optional)
  - Only check the clause if the guard is satisfied
  - Lambda function/Callable
  - Useful to implement state machines
- Events to match
  - port.RECEIVE(message)
  - port.RECEIVE()
  - timer.TIMEOUT
  - ptc.DONE
  - ptc.KILLED
- List of actions
  - A sequence of lambda functions
  - May finish with the Testerman keyword REPEAT
- Port queues
  - Ports not involved in the alt are untouched
  - Messages received during the alt on an involved port but not matched are dequeued/purged
  - The order matters: First message match, not best-match

TestCase

- The only entry points to an ATS
- Defined as a subclass of TestCase
  - class MyTestCase(TestCase):
  - body(self, vars) must be reimplemented
  - Automatically creates the MTC and system TC
    - self.mtc (or mtc ?)
    - self.system (or system ?)

- Run it from the Control Part
  - MyTestCase().execute()
  - MyTestCase("a description").execute(param1 = …)
  - Returns the TestCase verdict (=MTC verdict)

Behaviour

- A scenario runnable into a (P)TC
  - The TestCase.body definition is the scenario of the MTC
- Defined as a subclass of Behaviour
  - MyBehaviour(Behaviour)
  - body(self, vars) must be reimplemented
- Run it from when running a PTC
  - MyPtc.start(MyBehaviour(self))
  - MyPtc.start(MyBehaviour(self), param1= …)
  - (Warning: the testcase may not be a parameter after all)
- Wait for its completion
  - MyPtc.done()
- The PTC verdict automatically updates the MTC's on completion

Verdict Management

- A Verdict is associated to
  - MTC (TestCase verdict)
  - PTC (local verdict)
- The TestCase verdict is the combination of each TC verdicts

- Status
  - « none » (default)
  - « inconc » (inconclusive)
  - « pass »
  - « fail »
- Overriding precedence:
  - none < pass < inconc < fail

  - Set with self.setverdict()
    - from a TestCase body/MTC or Behaviour body/PTC
  - Retrieved with self.getverdict()

Best Practices

- Reusability rules
  - TestCases are completely described by themselves
    - They can be run individually
      - Well.. Ideally
    - The idea is that no code is executing in the Control section
      - Except the TestCase control itself
  - Do not update the Test Configuration (TTCN-3 speaking) once a PTC is started
    - i.e. do not create other PTC nor update connections once a PTC is started
  - Work on a Behavior library
    - Either message- or simulator oriented
    - Document the TC configuration they require
      - i.e. the port connections they need/expect
    - To reuse them
      - Create a PTC, connect it according to the behavior doc
      - Run the behavior within the PTC (using start())

Best Practices (2)

- A testcase shall not be used as a procedure or a function
  - Even if some TTCN-3 samples do it
- Use functions, TestCase, and behaviours carefully
  - Functions = serializable behaviour, tool; does not replace a testcase
    - Avoid creating a Testcase that boxes a single function
      - Less flexible that a testcase in case of testcase modification.
    - If your function needs an access to a probe/port, pass the TC (mtc or ptc)
      - it will run onto as first parameter
    - NEVER modify the TC status from a function
  - Testcase = targeted on what must be tested, with an associated OK/KO
    - Do NOT use a TestCase as a function (and avoid testcase status pollution)
  - Behaviour
    - Use it for parallelizable behaviours.
      - If the behaviour is always serialized, you may use functions instead.
    - It’s generaly more reusable to always create functions, then call them from a Behaviour

Coding Hints

- State machines implementation
  - State machine implementation in a Behaviour
  - Requires state management
  - Lambda functions cannot set external (immutable) variables

  - You may use a StateManager class:

```
def body(self, vars):
state = StateManager('waiting')
alt([
  [ lambda: state.get() in [ 'waiting' ],
    port.RECEIVE(template_INVITE()),
    lambda: state.set('incoming-call'),
    lambda: port.send(template_100Trying()),
    REPEAT,
],
])
```

Designing Script Parameters

- For both Campaign and ATS
- These parameters
  - Are adaptation variables
    - Renders your script suitable to be executed on any platform
  - Must NOT change the test purpose !

  - Do NOT turn your ATSes into a general purpose tool
    - If you want to use Testerman as a packet generator, you can
    - In this case, don’t call it a « test », but create a Testerman application instead

  - Must be
    - Enough parameters to be able to run your testcases on any platform
      - Probe names
      - SUT IP adresses
      - Non-guessable/discoverable object IDs, constraints, prerequisites

    - Minimal number of parameters to be able to run yout testcases on any platform
      - Don’t make generated IDs parametrized, but use a prefix instead
      - Try to discover everything you can discover automatically
      - Too many parameters will discourage those who will execute the test

Designing Script Parameters (2)

- Try to reuse parameters from one ATS to another one
  - Except if they are designed to be run in parallel
- Campaigns
  - (TODO) An option to gather parameters from all ATSes
    - No additional parameters to design
  - Transmitted to the ATSes
    - Conditionally, according to the branch

Prerequisites Management

- Shared prerequisites
  - « A, B, C users must be created with these properties »
  - Used for multiple ATSes or testcases
  - Let’s define it as a « test bed »
  - Can be automated
    - Dedicated ATS, called « pseudo ATS », containing « pseudo TestCase » to create it
      - This is not a « test », this is a provisioning/preparation tool
  - What cannot be automated must be described in the test bed « pseudo ATS » prerequisites
  - Integrated into a campaign
    - ats test_bed_creation.ats
      - ats my_ats_1.ats
      - ats my_ats_2.ats
      - ats my_ats_3.ats
    - ats test_bed_deletion.ats
- Embedded Preamble/Postamble execution for test cases and campaign is a Work In Progress

Prerequisites Management (2)

- Local prerequisites
  - A prerequisite for one testcase
    - Automate it when possible
      - Within the testcase itself
        - Functions are welcome
  - Automate its cancellation, too, even in case of a test error
  - Enable multiple executions of the same testcase without additional preparation
  - When no automation is possible
    - Document the prerequisites in the ATS « prerequisites » properties

Test Bed & ATS Parameters

- Since test bed creation is achieved through a pseudo-ATS (for now)
  - You need to choose correct ATS parameters to create it
    - Not too many
    - Enough to be adaptable

- Good way to see if your test bed creation is adaptable
  - Run it once
  - Don’t delete it
  - Change several parameters,
  - Run it with these new parameters
    - Should work. If not, not adaptable enough

  - Best practice: use a name prefix for all automatically created objects

High-level Behaviours

- Application-level behaviour to control low-level probes
  - We try to keep probes with low intelligence
  - So the complexity is sent up to the TTCN-3 world
  - Don’t pollute your actual testcase with it: use smart behaviours: simulators

- Simulators are not « oracles »
  - They won’t affect the testcase verdict, they don’t know « what we should expect »
  - The testcase (or an oracle behaviour) controls the simulator
    - Typically through a « control port »
  - The simulator can send feedback on interesting high level events

- Example of interesting simulators
  - Virtual Endpoints (VE)
    - Controlling VoIP signalling and RTP probes
    - Controlled by high-level control messages « send-call », « reject-call »
    - Implements a signaling and call stack
    - Send high-level control events « ringing », « ringback-tone »
  - Diameter RFC4006 server
    - Controlling a Diameter probe
    - Manage DWR/DWA in the background
    - Notifies of interesting messages (CCR), waiting for control feedback (« ok, continue », « drop the call », …)
  - HLR simulators, ...

Virtual Endpoint API

- Pending specification
- Will enable IPPhone vendors to implement reusable Testerman-based simulators