TTCN-3 User Conference 2008
Madrid, Spain

uOttawa

L'Université canadienne
Canada's university

A comparison between TTCN-3 and Python

by Bernard Stepien, Liam Peyton

School of Information Technology and Engineering

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Motivation

- Report on frequent remarks when presenting TTCN-3 to Industry in North America.
- Give an example for Python users where they can see how to translate a Python program into TTCN-3 and improve their testing.
- Promote adoption of TTCN-3 by highlighting the differences with Python.

Most common fears about TTCN-3

- Is a test specification language, not an Object Oriented language.
- Pattern matching with some regular expressions.
- Specialized test-focused community
- Cannot debug test cases.
- Writing codec is not trivial.
- Port communication concept misunderstood.

Known limitations of Python

- OO language but not strongly typed.
- Full regular expressions but with restricted set-based matching.
- Python has a general purpose programming community (not testing).
- No Eclipse plug-in.
- Cannot group test cases.
- Only two verdicts (pass or fail).
- By default no parallel test cases.

Misconceptions about Python as a test language

- Matching mechanism can be achieved using Python’s built-in structured equality operator “==“.
- Parallel test component can be implemented in Python with multi-threading.
- No learning curve, especially for the test adapter.

Central TTCN-3 concepts

- Data types.
- Composite events.
- Templates.
- Matching mechanism.
- Signature templates for procedures.
- Behavior trees
- Altstep
- Parallel test components.
- Separation of concerns.
- Operational semantics.

Python concepts

- Python is an interpreted language. Thus there is no compile time error detection.
- Python is a dynamic language. Nothing is really set in advance. Things can change at any time. This implies that if you change something accidentally, your test will no longer run or worse, interpret results incorrectly.
- Complex structural equality operator “==“

Important conceptual differences

- TTCN-3 has a clear model on how to structure a test.
- Python has no model. The user is left to his imagination. “pyUnit” is a poor solution too.
- With Python, the risk for bad design is great.
- With TTCN-3 the design will always follow the same model.
- The clear, unavoidable TTCN-3 model ensures exchangeability of test suites among players.

TTCN-3 Data types to Python

- Python has objects.
- Python objects don’t naturally have attributes.
- Python object attributes are not typed.
- Python objects attributes:
  - Are not declared explicitly.
  - are declared implicitly at initialization time in the object constructor (initializer),
  - in isolation anywhere in a program where the object instance is actually used.
- Python attributes declarations are dynamic.

- TTCN-3 data types are used for two purposes:
  - Normal variables definitions.
  - Template definitions.
- TTCN-3 Templates
  - full structured datatypes
  - parametrization to enable sophisticated matching
  - Dynamically instantiated into a strongly typed run-time variable
- TTCN-3 data types are used for strong type checking at design time.

TTCN-3 Composite events

- Oracles are specified around test events:
  - Messages being sent or received
  - Procedures being invoked or returning values.
- All the data gathered in a test event is processed at once using TTCN-3’s built-in matching mechanism.
- Thus, TTCN-3 can be described as **composite test event-centric**.

Templates and matching mechanism

TTCN-3 templates to Python

- TTCN-3 templates could be mapped to Python object instances.
- However, there are serious limitations using the above technique with Python.
- Python objects are practically typeless.

Templates differences

TTCN-3:

```
Type record myType {
    chartstring field_1,
    integer field_2,
    bitstring field_3
}

template myType myTemplate := {
   field_1 := “abc”,
   field_2 := 25,
   field_3 := ‘0110’
}
```

Python:

```
Class myClass:
   def __init__(self, theA, theB, theC):
        field_1 = theA
        field_2 = theB
        field_3 = theC
…
# create an object instance

myTemplate = myClass(‘abc’, 25, ‘0110’)
```

- In TTCN-3 templates, there is a direct connection (WYSIWYG) between field names and values.
- The TTCN-3 template is a one step feature.

- In Python, the class instantiation does not show the field names, thus, prone to errors, especially due to typelessness.
- The python template requires two independent steps.

Matching concepts

- TTCN-3 does bulk matching of all the elements of a data structure at once.
- Enables overview qualities of matching results
- Has optional fields.
- Has unlimited combinations of wildcard fields.

- Python would allow two different modes of matching:
  - Single element matching.
  - bulk matching using the “==“ operator on objects.
  - Has no optional fields
  - Has no flexible field based wildcards.

Matching: ttcn-3 vs python

- Strict values
- Alternate values
- Wildcards
- Optional fields
- Complex data structures
- Parametrization

Set differences

- In python, like in Java, sets do not allow duplicate elements.
- In TTCN-3, sets do allow duplicate elements.
- In fact, in TTCN-3 the set data type should really be called a **bag**.

Matching mechanism differences

- In python, the equality operator does not work with class instances.
- In python, the equality operator for classes has to be defined by the user in the class definition.
- In python classes, only one unique user defined equality operator can be defined.

- In TTCN-3, different templates with different matching rules can be specified using the same data type.
- the matching mechanism is fully built-in and does not need to be written by the user.

Python class matching example

```
class engine:
    def __init__(self, theNbPistons, theFuelType):
        self.nb_pistons = theNbPistons
        self.fuel_type = theFuelType

    def __eq__(self, other):
        return self.nb_pistons == other.nb_pistons \
              and self.fuel_type in other.fuel_type
…
assert aTemplate_1 == aTemplate_2
```

```
TTCN-3:
           …
           match(aTemplate_1, aTemplate2)
```

Handling of TTCN-3 wild cards

- In Python the TTCN-3 wildcards (*, ?) can only be implemented by either:
  - not specifying an equality for a field
  - By using regular expressions for a given field “(.*)”.
- This prevents the use of an object constructor (initializer) (__init__) to represent templates.
- Two different templates with different wild cards fields can only be represented by different constructors, not different object instances.
- In Python you can define only one constructor and one equality operator per defined class.

Wildcard Example

- In Python this can only be implemented using regular expressions in a fixed manner.

TTCN-3

```
template myType templ_1 := {
   field_1 := ?,
   field_2 := “abc”
}

template myType templ_2 := {
   field_1 := “xyz”,
   field_2 := ?
}
```

match(templ_1, templ_2) will succeed

Python

```
class MyType_1:
   def __init__(theA):
      field_1 = theA

class MyType_2:
   def __init__(theB):
      field_2 = theB

templ_1 = MyType_1( “abc”)

templ_2 = MyType_2(“xyz”)
```

templ_1 == templ_2
will be rejected in Python

Python objects limitations

- Only one constructor at a time allowed (no polymorphism allowed).
- If duplicate, only takes the second definition

```
class AClass:
   def __init__(self, theF1, theF2):
      self.af1 = theF1
      self.af2 = theF2

   def __init__(self, theF1):
      self.af1 = theF1
      self.af2 = 25

   def __eq__(self, other):
      return self.af1 == other.af1 \
        and self.af2 == other.af2

a1 = AClass('red', 4)
```

```
Traceback (most recent call last):
  File "multi_constructors.py", line 15,
                                          in <module>
    a1 = AClass('red', 4)
TypeError: __init__() takes exactly
                                 2 arguments (3 given)
```

Python objects limitations

- Only one equality operator allowed
- Only the second definition is used

```
class AClass:
    def __init__(self, theF1, theF2):
        self.af1 = theF1
        self.af2 = theF2

    def __eq__(self, other):
        print 'evaluating first eq'
        return self.af1 == other.af1 \
           and self.af2 == other.af2

    def __eq__(self, other):
        print 'evaluating second eq'
        return self.af1 == other.af1
…
assert a1 == a1
```

```
>>> ======= RESTART =========>>>
evaluating second eq
```

The first equality operator definition
Is merely ignored

Python object matching

obscure behaviors

- If two different python objects attributes names are identical (can happen by accident since Python is not strongly typed), then instances of these different objects can be compared.
- The above is inherently dangerous.
- However, this “trick” could be a solution to the previous wildcard combination problem.
- This is not a solution to the TTCN-3 behavior tree concept in Python.

Alternate values in templates
python

- Use the equality operator definition and the ‘in’ verb

```
class engine:
   …
    def __eq__(self, other):
        return self.nb_pistons == other.nb_pistons \
             and self.fuel_type in other.fuel_type
```

Problem: if you accidentally omit the list [‘gas’] in an object instance for a single element, the matching will no longer work and without warning.

```
anEngine1 = engine(6, 'gas')
anEngine2 = engine(6, ['gas', 'oil'])

assert anEngine1 == anEngine2   succeeds
But assert anEngine2 == anEngine1  fails
 File "C:\BSI_Projects\python\car_v1.py", line 8, in __eq__
    and self.fuel_type in other.fuel_type
TypeError: 'in <string>' requires string as left operand
```

Optional fields

- TTCN-3 can have optional fields

- Python does only strict matches:
  - Strict values
  - All fields must be present.

TTCN-3 template modifies feature

- Is unique to TTCN-3
- In Python, this would require writing an object duplication code.

```
template MyType myFirstTemplate := {
    field_1 := 5,
    field_2 := “done”,
    field_3 := { “a”, “f”, “g” }
}
```

```
template MyType myOtherTemplate modifies myFirstTemplate := {
     field_2 := “pending”
}
```

Template modifies in Python

- An assignment of an object to a new variable does not correspond to a duplication.
- New variable contains the instance of the previous variable.
- Modification of a field to the new variable propagates to the previous variable

```
engine_a = engine(6, 'gas')

print "engine_a: ", engine_a.fuel_type

engine_b = engine_a
engine_b.fuel_type = 'oil'

print "after modifying engine_b, engine_a: ", engine_a.fuel_type
```

```
>>> ==========RESTART ==========
>>>
engine_a:  gas
after modifying engine_b, engine_a:  oil
```

Differences in modifies feature

- In TTCN-3 the modifies creates a new copy of the referred template.
- The new template is as permanent and persistent as the one used to derive the new one.
- Once declared, a given template can no longer be modified.
- Only template parameters can modify a value on the fly.

- Python can modify the value of a field of an object instance any time with a reassignment.
- The problem with this is that the previous version of the instance object is no longer available.

Template pattern matching

TTCN-3

```
template MyType templ_1 := {
    str_field := pattern “abc*xyz”
}
```

Python

Using regular expressions?

Not with classes?

TTCN-3 vs Python regular expressions

- TTCN-3 specifies regular expressions in the template field.
- Thus, two different templates can have two different regular expressions for the same field.

- Python can only specify the regular expression in the user defined equality operator __eq__
- Thus a regular expression for a given field can be defined only once.

Behavior trees

TTCN-3 behavior tree to Python

- A TTCN-3 behavior tree can be represented with a collection of nested if-then-else constructs in Python.
- TTCN-3 however, with the combination of behavior trees and templates achieves one important separation of concern:

  Separating behavior from conditions governing behavior.

TTCN-3 behavior tree features

- The TTCN-3 behavior tree is based on TTCN-3 snapshot semantics.
- Representing a behavior tree with a python nested if-then-else is not always possible when there is more than one data type for received values.

TTCN-3 behavior tree concept

- Each branch can operate on different data types.
- Each branch can operate on different ports
- TTCN-3 has an implicit and transparent message queue look up.
- TTCN-3 behavior tree is a very concise notation.

Behavior tree in Python

- Does not work because different objects of different classes can not be compared.
- Must use an isinstance(…) construct.

```
class AClass:
    def __init__(self, theF1, theF2):
        self.af1 = theF1
        self.af2 = theF2
 def __eq__(self, other):
        return self.af1 == other.af1 \
           and self.af2 == other.af2
```

```
class BClass:
    def __init__(self, theF1, theF2):
        self.bf1 = theF1
        self.bf2 = theF2
    def __eq__(self, other):
        return self.bf1 == other.bf1 \
           and self.bf2 == other.bf2
```

```
a1 = AClass('red', 4)
b2 = BClass('red', 4)                    a1 == b2
```

```
Traceback (most recent call last):
  File "C:/BSI_Projects/python/object_matching", line 41, in <module>
    if a1 == b2:
  File "C:/BSI_Projects/python/object_matching", line 8, in __eq__
    return self.af1 == other.af1 \
AttributeError: BClass instance has no attribute 'af1'
```

Behavior tree in TTCN-3

- TTCN-3 uses the concept of separation of concerns between the abstract layer and the adaptation layer.
- The actual decoded or encoded values are stored in an invisible variable.
- If the type of the decoded variable does not match, TTCN-3 moves on to the next condition without raising an error until it matches the right type and then the right value. Thus, the isinstance(…) checking is implicit.
- Conclusion: the TTCN-3 behavior tree is more than nested if-then-else constructs.
- Another non visible aspect is the snapshot mechanism that is part of execution tools (no programming effort).

Behavior tree example TTCN-3

```
type record typeA {
   integer A_field_1,
   charstring A_field_2
}

type record typeB {
   charstring B_field_1,
   integer B_field_2
}
```

```
type port APortType message {
   in typeA;
   out charstring;
}

type port BPortType message {
   in typeB;
   out charstring;
}
```

```
template typeA templateA := {
    A_field_1 := 58,
    A_field_2 := "abcd"
}

template typeB templateB := {
    B_field_1 := "xyz",
    B_field_2 := 279
}
```

```
testcase myMultiTypeTest() runs on MTCType {
   … map(…) // for both ports
   portA.send("request A");
   portB.send("request B");

   interleave {
      [] portA.receive(templateA) { }
      [] portB.receive(templateB) { }
   }
   setverdict(pass)
}
```

Behavior tree example Python

```
def myMultiTypeTestCase():

    portA = map("portA", 9000)
    portB = map("portB", 9001)

    send("portA", 'getKind\n\r')
    send("portB", 'getHeight\n\r')

    if receive("portA", templateA):
        if receive("portB", templateB):
            print 'verdict pass'
        else:
            print 'did not receive templateB - verdict fail'
    elif receive("portB", templateB):
        if receive("portA", templateA):
            print 'verdict pass'
        else:
            print 'did not receive templateA - verdict fail'
    else:
        print 'receive unexpected templateA – v fail
```

- The corresponding TTCN-3 map, send and receive statements must be custom written in Python.
- The interleave construct must be hand written.

Send in Python

user custom written code

```
def send(portName, template):
    port = connections[portName]
    # TODO write an encode

    port.send(template)

    e = event('send', portName, template)
    events.append(e)
```

Receive in Python

user custom written code

```
def receive(port, template):
    receivedSomething = False
    while receivedSomething == False:
        try:
            in_msg = msgQueue[port]
            receivedSomething = True
        except KeyError:
            receivedSomething = False

    if isinstance(in_msg, typeA) and in_msg == template:
        print 'in receive template matched'
        e = event('receive', port, template)
        events.append(e)
        return True
    elif isinstance(in_msg, typeA) and …
    else:
        print 'did not receive template - verdict fail'
        return False
```

- Two parts:
  - – Retrieving data from the port.
  - – Matching the incoming message to the template.

TTCN-3 altstep concept

- Is basically defined only as a macro.
- But it is more than a macro because of the underlying TTCN-3 snapshot semantics.
- Thus it is a rather very powerful structuring concept
- Enables factoring out behavior.
- What is factored out is a sub-tree, including the conditions governing behavior.
- Since Python does not support macros, this would be impossible to implement except through custom code.
- Not obvious to implement in Python due to the underlying TTCN-3 snapshot semantics.

altstep example

TTCN-3

```
alt {
   [] p.receive(“a”) { …}
   [] p.receive(“b”) { …}
   [] p.receive(“c”) { …}
}
```

```
alt {
   [] p.receive(“a”) { … }
   [] other_behavior()
}

altstep other_behavior() {
   alt {
        [] p.receive(“b”) { …}
        [] p.receive(“c”) { …}
   }
}
```

Python

```
If response == “a”:
              …
else if response == “b”:
              …
else if response == “c”:
              …
```

- Python can not split an if-then-else into functions, nor does it support macros.
- At best it could factor out the body of an if or an else, but not the condition.
- This is because a python function invocation is only a sequential construct.

Matching differences
TTCN-3 match vs Python assert

- TTCN-3 has the concept of matching (implicit in receive, or explicit with the keyword match)
- In a behavior tree, the match is not final (not an assert). It will look up the specified alternatives until a match is found.

- In Python, the assert statement produces a failure if no match and stops execution at the point of failure.
- The assert is not capable of doing an if-then-else or a case statement.

Python assert

- The Python assert feature is a one try only feature.
- If the assert fails, the program stops.
- Thus, the assert can not be used to simulate the TTCN-3 like behavior tree.
- In TTCN-3, if an alternative does not match, the next alternative is tried.
- The weakness of the assert is not Python specific, other general programming languages have the same problem (Java/JUnit, etc…)

Programming styles considerations

Python single element matching

- There are three activities in testing:
  - Obtaining data over a communication channel
  - Parsing data to extract the relevant field
  - Matching the extracted data to an oracle.
- With general programming languages like Python, users have a tendency to cluster or intertwine the three above activities together in their code.
- This structuring habit leads to poor maintainability.

Python single element testing example

- Three steps of testing:
  - Step 1: Obtain response from SUT by reading data from a communication channel.
  - Step 2: Parse the response data to extract a single piece of data.
  - Step 3: Make assertions on single pieces of data.
- In the absence of a model, testers have a tendency to cluster the above three steps together.

Establish a connection:

```
connection = socket.socket(…)
connection.connect((HOST, PORT))
```

Step 1: read response data:

```
response = connection.recv(1024)

# received response is: 'joe   178‘

name = response[0:5]           # step 2
assert  name.strip() == 'joe‘  # step 3

height = int(response[6:9])    # step 2
assert  height == 178          # step 3
```

Programming styles

- The TTCN-3 template concept enforces a style whereas the tester must consider all the elements of a problem in a single matching operation.
- Python can most of the time do the same as ttcn-3 but tends to encourage a style where elements of a problem are handled one at a time.

pyUnit
unittest library

- Is merely JUnit implemented using python
- Limited functionality due to unit testing approach.
- No easy solution for parallelism.
- Not even the limited GUI functions of JUnit.

Strong typing

Strong typing differences
Python

```
“ottawa” == 12345
```

- In python:
  - will not match. Clear!
  - But it is a silent error.
  - It will reveal itself only at run time rather than at compile time during development.

Typing errors
python object instances

```
anEngine3 = engine(6, 'gas')
anEngine4 = engine('gas', 6)

assert anEngine3 == anEngine4
```

The above will fail silently or without explanations

Here, the error is due to the lack of typing
that could have warned the user about
the accidental permutation of values.

Strong typing differences
TTCN-3

```
type record someinfoType {
    charstring city
}
```

```
template someinfoType myInfoTemplate := {
            city := 12345
}
```

- In TTCN-3, this would raise an error at compile time.
- Why is this important?:
  - A test suite may be used by many testers.
  - A silent error like in Python is thus spread among users, some will detect it, some won’t.

Python objects dynamic attribute names issue

- Not really a typing issue.
- But with the same consequences.

```
class myClass:
    def __init__(self, theColor, theSize):
        self.color = theColor
        self.size = theSize
    def __eq__ …

a = myClass(‘red’, 10)
b = myClass(‘blue’, 15)

assert a == b    # will not match, OK
```

Down the road:

```
a.colour = ‘blue’
a.size = 15

a == b # should have matched
              but will still not match
               but for the wrong
              reason
```

Parametrization

- TTCN-3 allows parametrization of:
  - Templates
  - Functions
  - Test cases
  - Parallel test components

- Python allows parametrization of
  - Templates with serious limitations
  - Functions
  - Threads with limitations

Parallel test components

Parallel test components

- **TTCN-3**
- Creation
- Variables
- Communication
- Behavior parameter
- Test component coordination

- **Python**
- Multi-threading class extension
- Creation
- Use of class functions
- Communication

Parallel test components –TTCN-3

```
type component ptcType {
   port networkPortType network;
}

testcase phoneSystemTest() runs on MTCType {
   var ptcType user[2];

   user[0] := ptcType.create;
   user[1] := ptcType.create;

   user[0].start(user_1_behavior("555-1212"));
   user[1].start(user_2_behavior(“911"));

   all component.done;

   log("testcase phoneSystemTest completed");
}
```

- Concise test case.
- Creation using types
- Starting by indicating which test behavior to use.
- Enables parametrization of behavior at start time.

Parallel test components – Python

```
import threading

class Ptc(threading.Thread):
   def __init__(self, name, number):
      threading.Thread.__init__(self)
      self.name = name
      self.number = number

   def user_behavior_1(self, req_number):
      print 'user 1 is requesting number: ', req_number
   def user_behavior_2(self, req_number):
      print 'user 2 is requesting number: ', req_number

   def run(self):
      print 'starting PTC for', self.name

      if self.name == 'user_1':
         self.user_behavior_1(self.number)
      elif self.name == 'user_2':
         self.user_behavior_2(self.number)

user = []
user.append(Ptc('user_1', '555-1212'))
user.append(Ptc('user_2', '911'))

user[0].start()
user[1].start()…
```

- Need to write a special multi-threading class extension.
- No way to communicate directly which function to run at start time.
- External functions do not have access to the Ptc object instance attributes.
- Poor object intercommunication
- Parametrization is limited due to the restriction of only one initializer allowed.
- There really is no polymorphism.

Tracing, debugging, matching results inspection

Displaying results

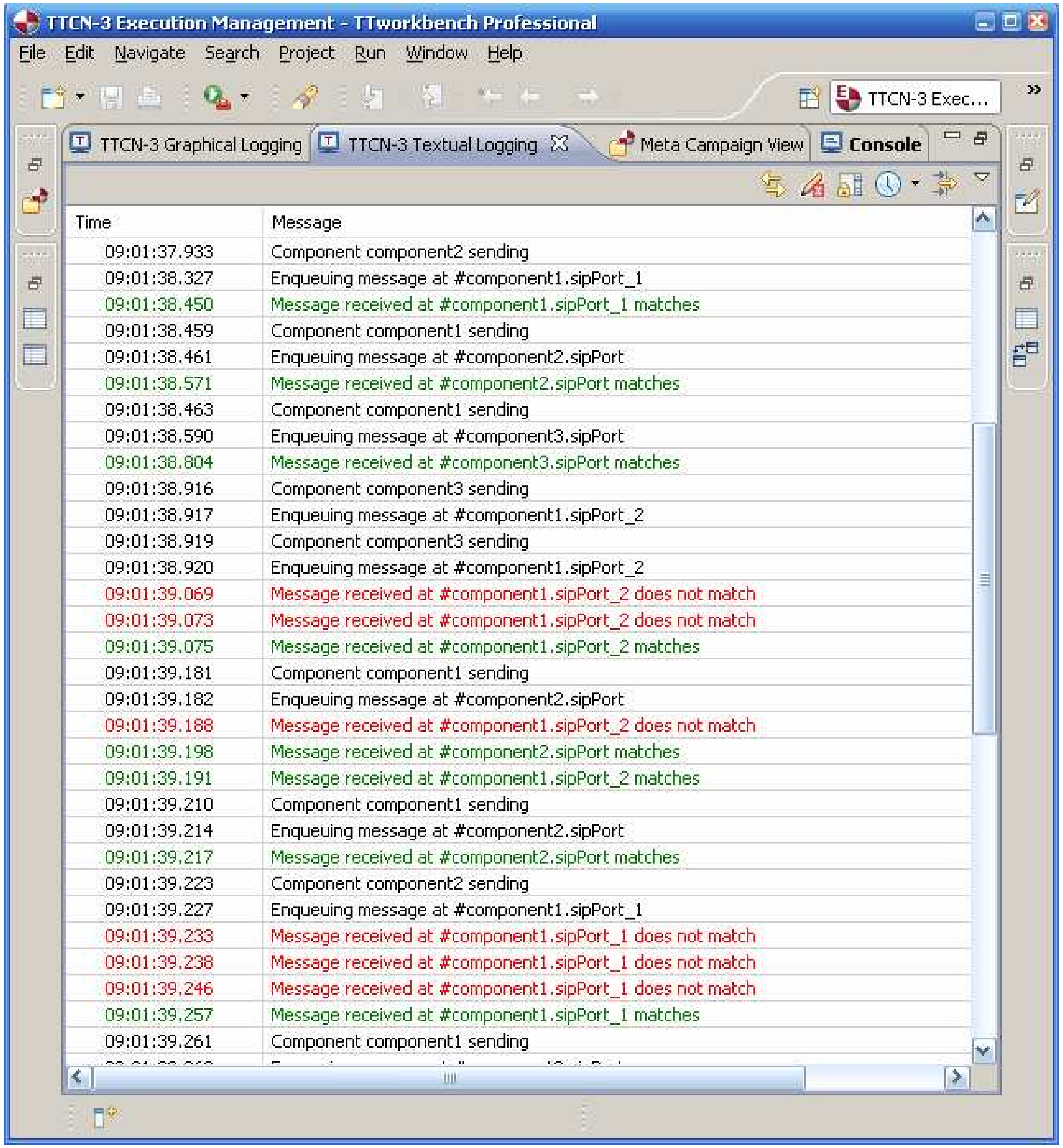
- In **TTCN-3**
- The tools provide:
  - Test events inspection
  - Tracing based on test events

- In **Python**:
- You must develop your own custom events lookup functions
- Tracing is function invocation based.

Composite TTCN-3 test event tools inspection facilities

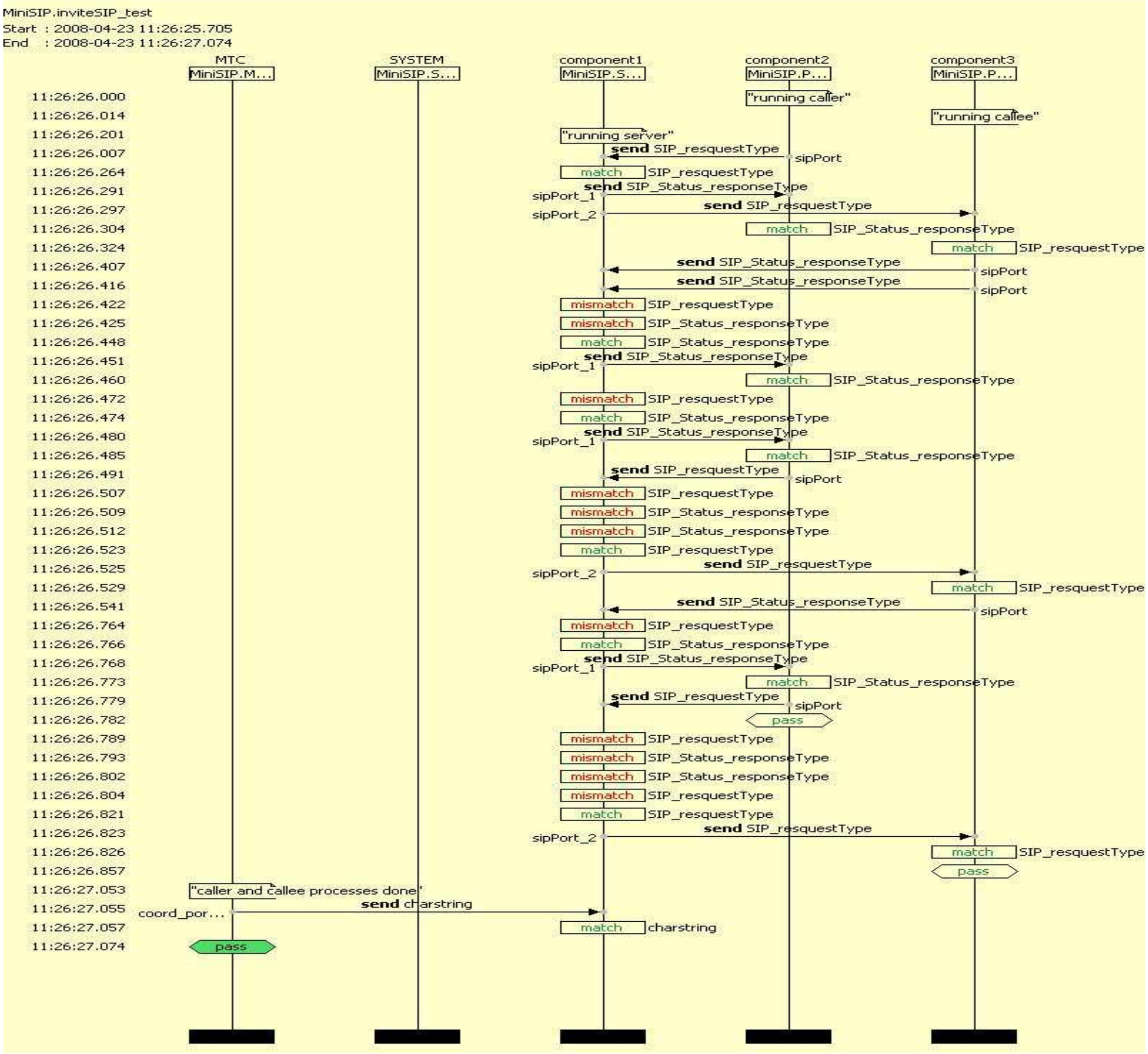
- Because the TTCN-3 model uses the concept of composite test event, tracing and debugging are fully centered around the composite event:
  - Composite event tracing.
  - Composite event matching results.
  - Composite event code location (context analysis).
- All tracing and lookup facilities are provided by the TTCN-3 tools. (no programming effort).

TTCN-3 composite event tracing



Time	Message
09:01:37.933	Component component2 sending
09:01:38.327	Enqueuing message at #component1.sipPort_1
09:01:38.450	Message received at #component1.sipPort_1 matches
09:01:38.459	Component component1 sending
09:01:38.461	Enqueuing message at #component2.sipPort
09:01:38.571	Message received at #component2.sipPort matches
09:01:38.463	Component component1 sending
09:01:38.590	Enqueuing message at #component3.sipPort
09:01:38.804	Message received at #component3.sipPort matches
09:01:38.916	Component component3 sending
09:01:38.917	Enqueuing message at #component1.sipPort_2
09:01:38.919	Component component3 sending
09:01:38.920	Enqueuing message at #component1.sipPort_2
09:01:39.069	Message received at #component1.sipPort_2 does not match
09:01:39.073	Message received at #component1.sipPort_2 does not match
09:01:39.075	Message received at #component1.sipPort_2 matches
09:01:39.181	Component component1 sending
09:01:39.182	Enqueuing message at #component2.sipPort
09:01:39.188	Message received at #component1.sipPort_2 does not match
09:01:39.198	Message received at #component2.sipPort matches
09:01:39.191	Message received at #component1.sipPort_2 matches
09:01:39.210	Component component1 sending
09:01:39.214	Enqueuing message at #component2.sipPort
09:01:39.217	Message received at #component2.sipPort matches
09:01:39.223	Component component2 sending
09:01:39.227	Enqueuing message at #component1.sipPort_1
09:01:39.233	Message received at #component1.sipPort_1 does not match
09:01:39.238	Message received at #component1.sipPort_1 does not match
09:01:39.246	Message received at #component1.sipPort_1 does not match
09:01:39.257	Message received at #component1.sipPort_1 matches
09:01:39.261	Component component1 sending

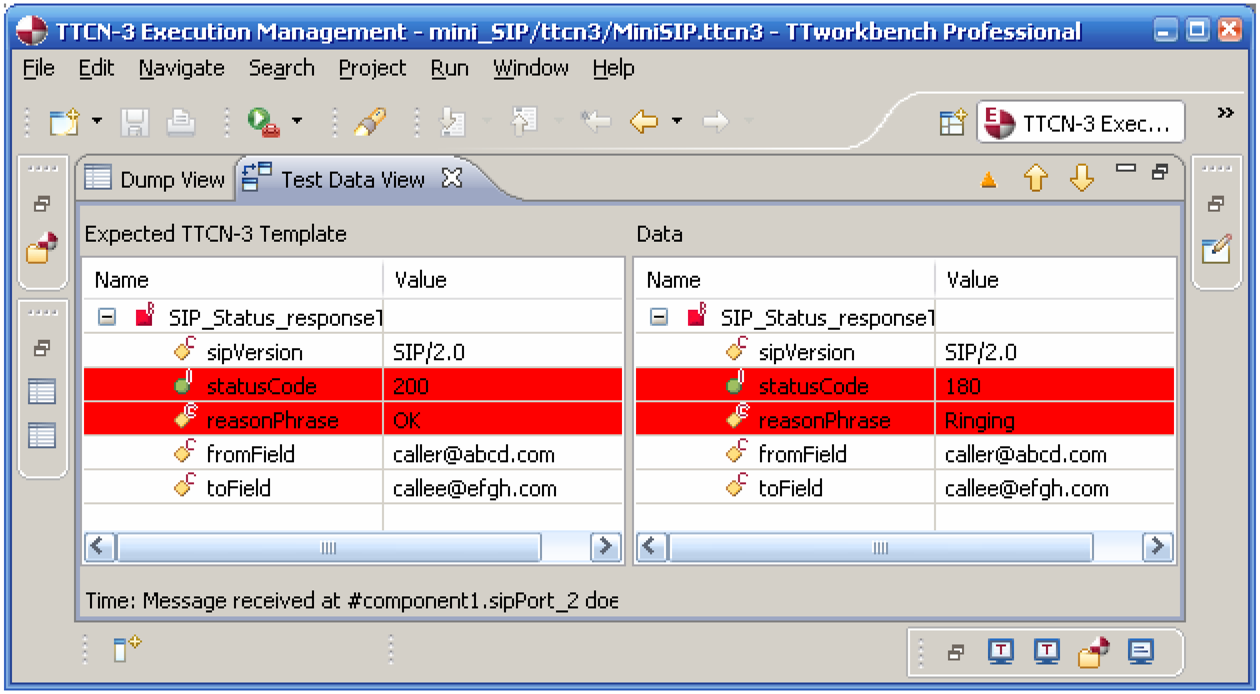
TTCN-3 graphic event tracing



```
MiniSIP.inviteSIP_test
Start  : 2008-04-23 11:26:25.705
End    : 2008-04-23 11:26:27.074
```

```
11:26:26.000  "running caller"
11:26:26.014  "running callee"
11:26:26.201  "running server"
              send SIP_resquestType
11:26:26.007  sipPort
11:26:26.264  match SIP_resquestType
              send SIP_Status_responseType
11:26:26.291  sipPort_1
              send SIP_resquestType
11:26:26.297  sipPort_2
11:26:26.304  match SIP_Status_responseType
11:26:26.324  match SIP_resquestType
              send SIP_Status_responseType
11:26:26.407  sipPort
              send SIP_Status_responseType
11:26:26.416  sipPort
11:26:26.422  mismatch SIP_resquestType
11:26:26.425  mismatch SIP_Status_responseType
11:26:26.448  match SIP_Status_responseType
              send SIP_Status_responseType
11:26:26.451  sipPort_1
11:26:26.460  match SIP_Status_responseType
11:26:26.472  mismatch SIP_resquestType
11:26:26.474  match SIP_Status_responseType
              send SIP_Status_responseType
11:26:26.480  sipPort_1
11:26:26.485  match SIP_Status_responseType
              send SIP_resquestType
11:26:26.491  sipPort
11:26:26.507  mismatch SIP_resquestType
11:26:26.509  mismatch SIP_Status_responseType
11:26:26.512  mismatch SIP_Status_responseType
11:26:26.523  match SIP_resquestType
              send SIP_resquestType
11:26:26.525  sipPort_2
11:26:26.529  match SIP_resquestType
              send SIP_Status_responseType
11:26:26.541  sipPort
11:26:26.764  mismatch SIP_resquestType
11:26:26.766  match SIP_Status_responseType
              send SIP_Status_responseType
11:26:26.768  sipPort_1
11:26:26.773  match SIP_Status_responseType
              send SIP_resquestType
11:26:26.779  sipPort
11:26:26.782  pass
11:26:26.789  mismatch SIP_resquestType
11:26:26.793  mismatch SIP_Status_responseType
11:26:26.802  mismatch SIP_Status_responseType
11:26:26.804  mismatch SIP_resquestType
11:26:26.821  match SIP_resquestType
              send SIP_resquestType
11:26:26.823  sipPort_2
11:26:26.826  match SIP_resquestType
11:26:26.857  pass
11:26:27.053  "caller and callee processes done"
              send charstring
11:26:27.055  coord_por...
11:26:27.057  match charstring
11:26:27.074  pass
```

TTCN-3 composite event matching results lookup


TTCN-3 Execution Management - mini_SIP/ttcn3/MiniSIP.ttcn3 - TTworkbench Professional
File
Edit
Navigate
Search
Project
Run
Window
Help
TTCN-3 Exec...
Dump View
Test Data View

Expected TTCN-3 Template

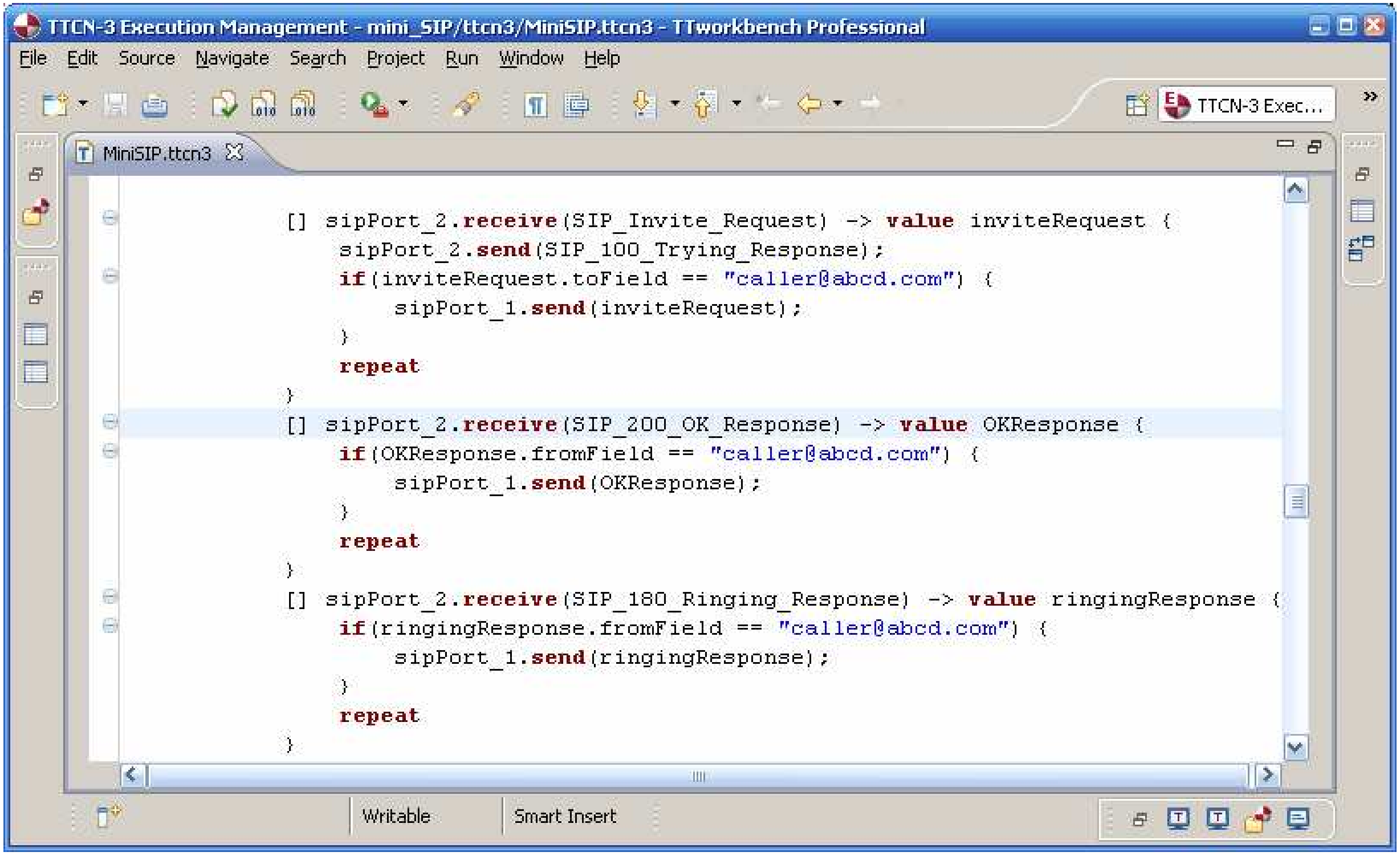
Name	Value
⊟ SIP_Status_response1	
sipVersion	SIP/2.0
statusCode	200
reasonPhrase	OK
fromField	caller@abcd.com
toField	callee@efgh.com

Data

Name	Value
⊟ SIP_Status_response1	
sipVersion	SIP/2.0
statusCode	180
reasonPhrase	Ringing
fromField	caller@abcd.com
toField	callee@efgh.com

Time: Message received at #component1.sipPort_2 doe

TTCN-3 composite event code locator



TTCN-3 Execution Management - mini_SIP/ttcn3/MiniSIP.ttcn3 - TTworkbench Professional
File Edit Source Navigate Search Project Run Window Help
TTCN-3 Exec...
MiniSIP.ttcn3
Writable
Smart Insert

```
[] sipPort_2.receive(SIP_Invite_Request) -> value inviteRequest {
   sipPort_2.send(SIP_100_Trying_Response);
   if(inviteRequest.toField == "caller@abcd.com") {
       sipPort_1.send(inviteRequest);
   }
   repeat
}
[] sipPort_2.receive(SIP_200_OK_Response) -> value OKResponse {
   if(OKResponse.fromField == "caller@abcd.com") {
       sipPort_1.send(OKResponse);
   }
   repeat
}
[] sipPort_2.receive(SIP_180_Ringing_Response) -> value ringingResponse {
   if(ringingResponse.fromField == "caller@abcd.com") {
       sipPort_1.send(ringingResponse);
   }
   repeat
}
```

Tracing with Python

- Produces only function invocation traces.
- No traces from successful test event matching path that leads to the point of error.

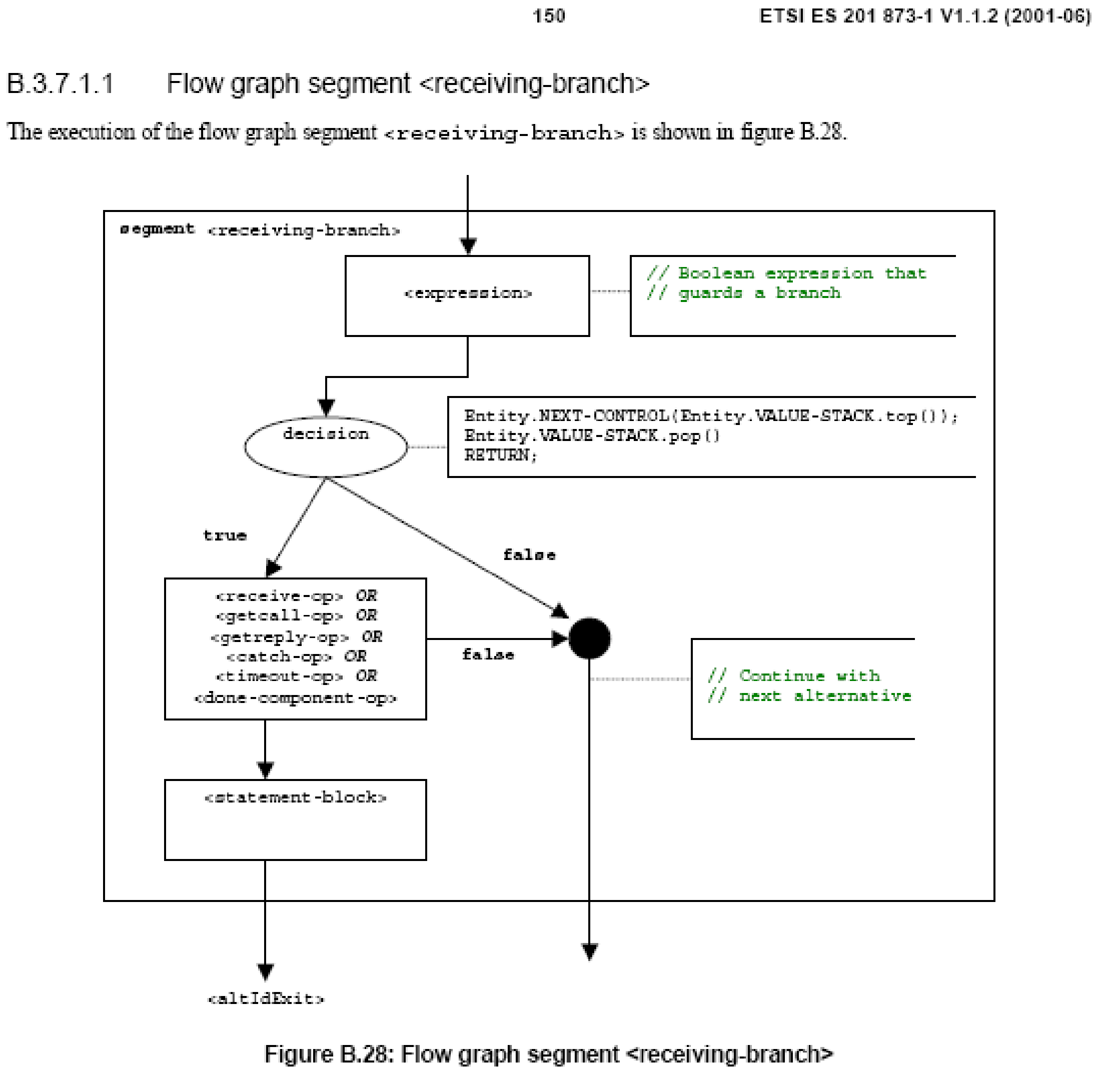
```
Traceback (most recent call last):
  File "C:/BSI_Projects/python/Behavior Tree Example/behaviorTreeEx_3.py",
              line 186, in <module> myMultiTypeTestCase()
  File "C:/BSI_Projects/python/Behavior Tree Example/behaviorTreeEx_3.py",
              line 170, in myMultiTypeTestCase     if receive("portB", templateB):
  File "C:/BSI_Projects/python/Behavior Tree Example/behaviorTreeEx_3.py",
              line 127, in receive    assert in_msg == template
AssertionError
```

TTCN-3 operational semantics

- Are defined as flow diagrams.
- Macros.
- Matching of messages
- There are 112 pages about operational semantics in the standard.
- The most important concept is that in TTCN-3, the operational semantics are clearly defined.
- In a Python program, nobody except the developer knows what the semantics are unless he has written a documentation (a rare fact).
- And because interpreted, weakly typed weird unintended behavior is possible outside the intended semantics when a “wrongly” typed input happens

TTCN-3 Operational semantics

flow diagram example



B.3.7.1.1 Flow graph segment <receiving-branch>

The execution of the flow graph segment <receiving-branch> is shown in figure B.28.

```
Entity.NEXT-CONTROL(Entity.VALUE-STACK.top());
Entity.VALUE-STACK.pop()
RETURN;
```

Figure B.28: Flow graph segment <receiving-branch>

Operational semantics

very detailed description in standard

B.3.4.5 Matching of messages, procedure calls, replies and exceptions

The operations for receiving a message, a procedure call, a reply to a procedure call or an exception are **receive**, **getcall**, **getreply** and **catch**.

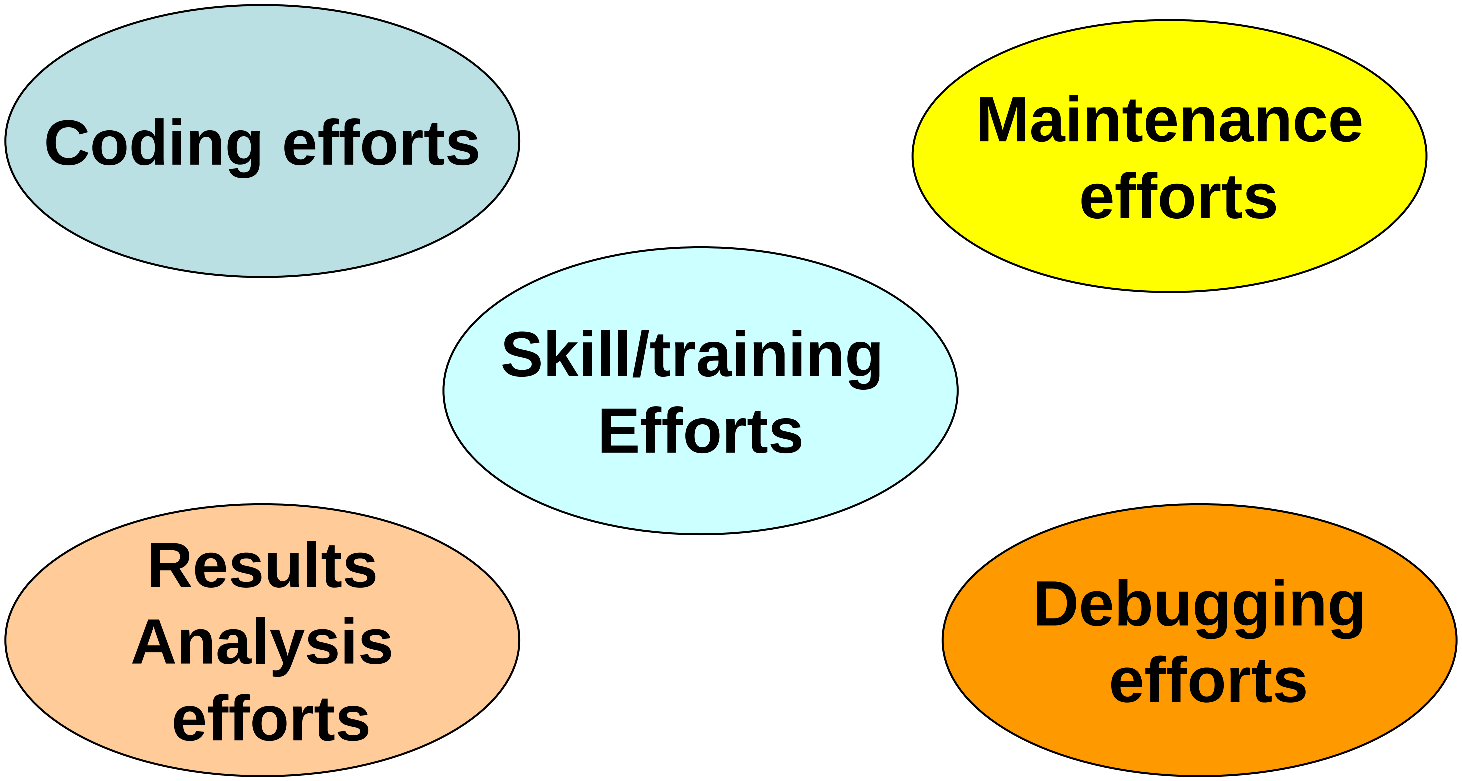
All these receiving operations are built up in the same manner:

```
<port-name>.<receiving-operation>(<matching-part>)
                                        [from <sender>] [<assignment-part>]
```

The <*port-name*> and <*receiving-operation*> define port and operation used for the reception of an item. In case of one-to-many connections a **from**-clause can be used to select a specific sender entity <*sender*>. The item to be received has to fulfil the conditions specified in the <*matching-part*>, i.e., it has to match. The <*matching-part*> may use concrete values, template references, variable values, constants, expressions, functions, etc. to specify the matching conditions. The operational semantics assumes that there exists a generic *MATCH-ITEM* function: *MATCH-ITEM(<item-to-check>, <matching-part>, <sender>)* returns *true* if <*item-to-check*> fulfils the conditions of <*matching-part*> and if <*item-to-check*> has been sent by<*sender*>, otherwise it returns *false*.

Cost benefit analysis

Cost factors



Coding efforts

Maintenance efforts

Skill/training Efforts

Results Analysis efforts

Debugging efforts

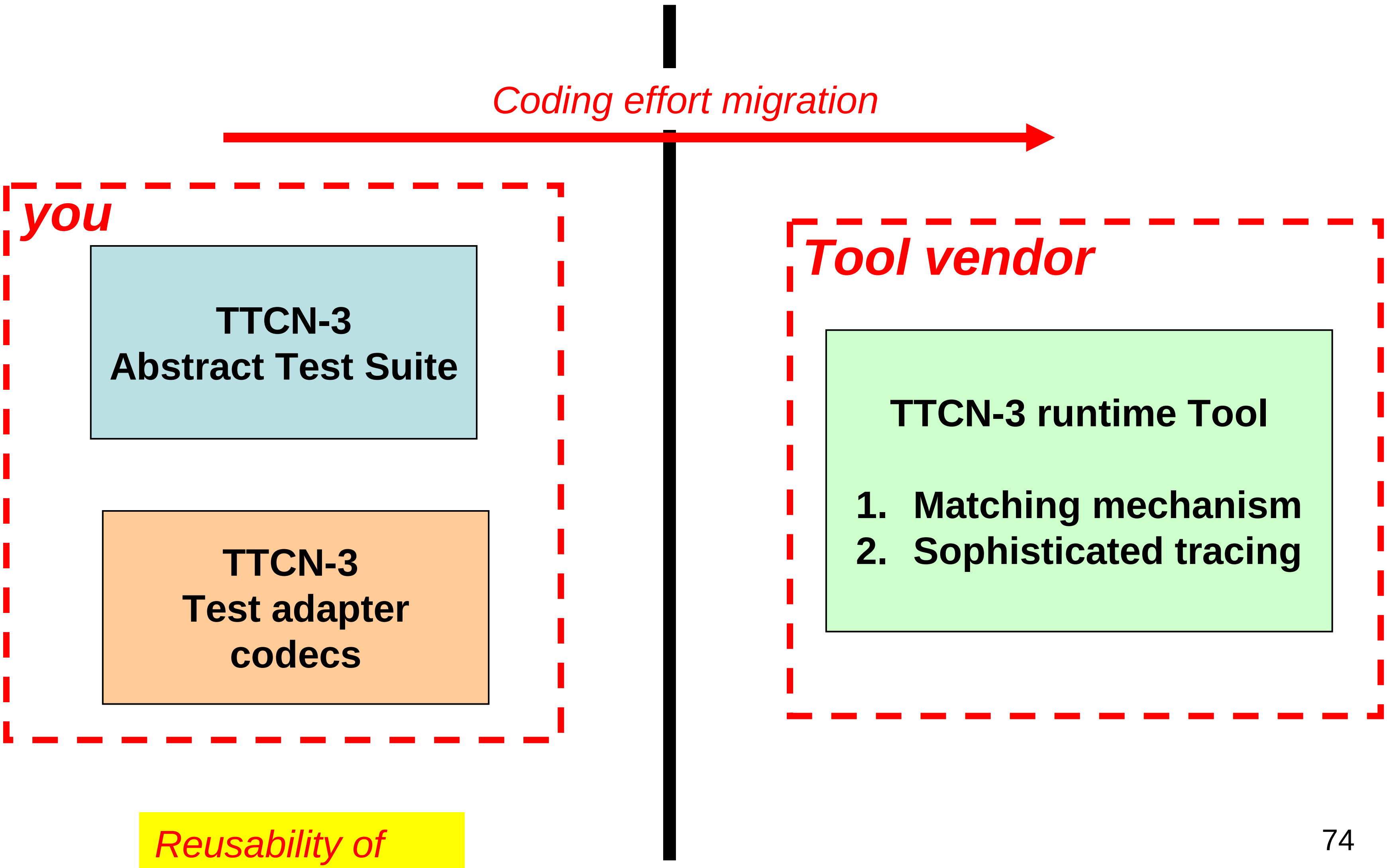
Coding efforts

Abstract Test Suite (ATS)

Test Adapter codecs

- TTCN-3 language constructs result in code reduction.
- The concept of Abstract Test Suite makes a test suite usable on different platforms or tools from different vendors.

TTCN-3 Coding efforts distribution



Coding effort migration

you

TTCN-3
Abstract Test Suite

TTCN-3
Test adapter
codecs

Tool vendor

TTCN-3 runtime Tool

1. **Matching mechanism**
2. **Sophisticated tracing**

Reusability of adapter codecs?

Coding effort reduction

- TTCN-3 provides coding effort reduction via:
  - Powerful language constructs and short hands.
  - Strong typing.
  - Strong semantics.
  - Test event centric tracing facilities.

- Python provides coding effort reduction via:
  - Typelessness (no datatypes to define).
  - Structured equality operator (not really true).
- Unfortunately, Python’s coding effort reduction mostly results in:
  - additional debugging efforts (resulting from typelessness).
  - Custom code development (structured equality operator, …).

Results analysis efforts

- Question: when a test has executed, how do we know that the results are correct?
- A pass verdict does not guarantee that the test is correct.
- Consequently, the easier it will be to analyze test results, the more you will trust these results.

Maintenance efforts

- TTCN-3’s central concept of separation of concern reduces maintenance efforts:
  - Concept of composite test event enables to zero in on a specific test functionality.
  - The separation between abstract and concrete layers code enables to zero in on specific test aspects (protocol, communications, codecs).
  - The risk of errors is reduced because there is less code to look at for a given functionality.

Debugging efforts

- Debugging efforts are directly linked to results analysis efforts.
- The composite test event centric aspect of TTCN-3 enables:
  - Zero in on an event.
  - Appraise the protocol part (ATS).
  - Appraise the test adapter and codecs.
  - Question the correctness of the tool used.
  - Question the TTCN-3 standard.

Skill/training efforts

the problematic

- **The myth:**
  - TTCN-3 has an extremely steep learning curve.
- **The reality:**
  - TTCN-3 is a large language
  - TTCN-3 has unusual but very powerful concepts
    - Templates
    - Matching mechanism
    - Behavior tree
    - Adaptation layer
  - There is no reason to start with a sub-set of the language where the learning curve is extremely shallow.
  - As skills build up, more powerful features can be learned.

Skill/training efforts

the help

- Where to learn TTCN-3?
  - ETSI web site tutorials.
  - Universities
  - Vendor’s courses
- Training cycle
  - Basic elements can be learned in a day.
  - Powerful features in three days.

Software economics

Factor	TTCN-3	Python
Skill/training	High	Low
Coding effort	Low	Low
Debugging effort	Low	High
Results analysis efforts	Low	High
Maintenance efforts	Low	High

Coding efforts details

Factor	TTCN-3	Python
Test event specification	Low	High
Test adapter	High	Low
Codec	High	Low
Display of results	**Nil**	high

Where to get TTCN-3 help?

- http://www.ttcn-3.org
  - standards
  - Tutorials
  - Papers
- http://www.site.uottawa.ca/~bernard/ttcn.html
  - Tutorials
  - Papers
  - Case studies

Conclusion

TTCN-3 is better than a general programming language

- Because it separates abstraction from implementation details.
- The template concept is considerably more flexible.
- Strong typing allows early detection of errors.
- The behavior tree allows better test behavior overview.
- Python is however an excellent choice for the test adapter and codecs and thus could be combined with TTCN-3.

Contact information

- bernard@site.uottawa.ca
- lpeyton@site.uottawa.ca
- http://www.site.uottawa.ca/~bernard/ttcn.html
- http://www.site.uottawa.ca/~lpeyton/