TTCN-3 Users conference Asia 2007
Beijing, China

Telelogic

Telelogic Tester™

Managing Concurrency and Parallel Testing with TTCN-3

Pierre Bentkowski,
Principal Consultant

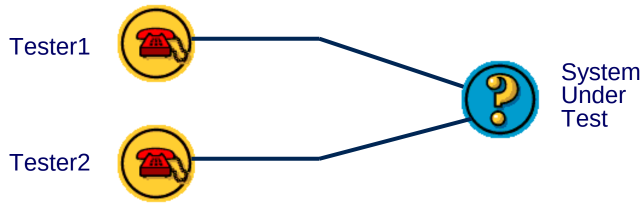
Telelogic

Concurrent TTCN-3

- Why do we need a concurrent test architecture?
- What kind of architectures can be used?
- How TTCN-3 supports such architectures?
- A TTCN-3 example
- TTCN-3 Configuration Operations
- Tips and Guidelines

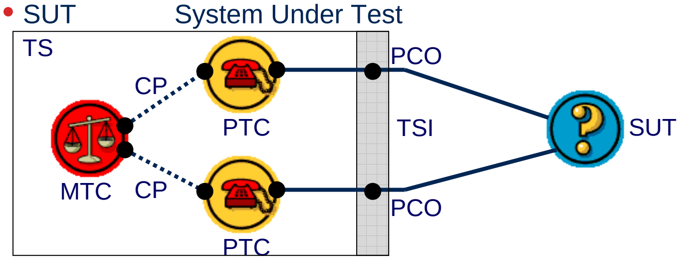
Terminology

- PCO Point of Control and Observation (Port type)
- CP Control Point (Port type)
- MTC Main Test Component (Component type)
- PTC Parallel Test Component (Component type)
- TS Test System
- TSI Test System Interface
- SUT System Under Test



Terminology

- PCO Point of Control and Observation (Port type)
- CP Control Point (Port type)
- MTC Main Test Component (Component Type)
- PTC Parallel Test Component (Component Type)
- TS Test System
- TSI Test System Interface
- SUT System Under Test

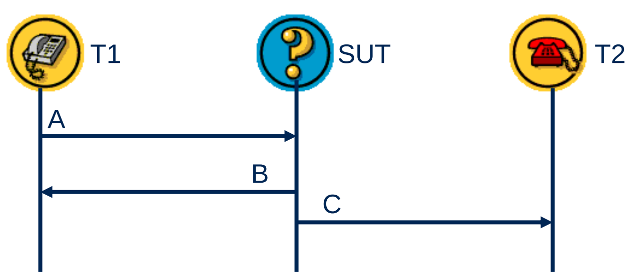


Why do we need a concurrent test architecture?

- By nature, devices and users which are interfaced to the SUT are functioning in a concurrent manner.
- Even with perfectly synchronized inputs to the SUT, there are no guaranties that the SUT will reply with the exact same sequence of outputs.

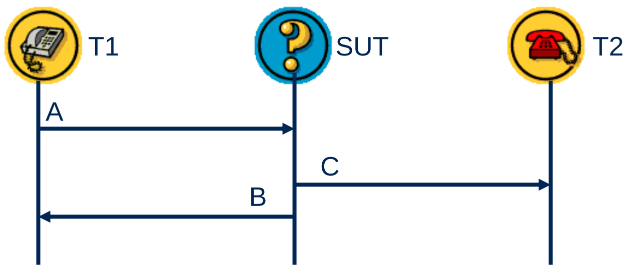
Why do we need a concurrent test architecture?

- Tester T1 sends the message A to the SUT
- The SUT replies to both Testers, T1 and T2
- The SUT first sends B to T1, then C to T2
- Therefore the Test Sequence is {T1!A,T1?B,T2?C}



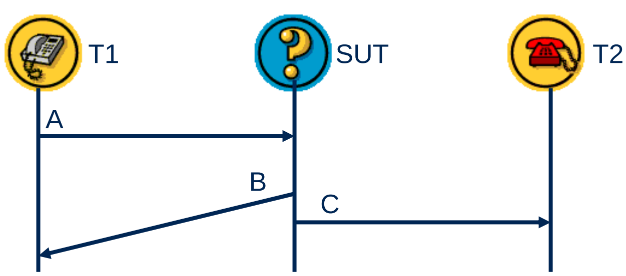
Why do we need a concurrent test architecture?

- Tester T1 sends the message A to the SUT
- The SUT replies to both Testers, T1 and T2
- The SUT first sends C to T2, then B to T1
- Therefore the Test Sequence is {T1!A,T2?C,T1?B}



Why do we need a concurrent test architecture?

- Tester T1 sends the message A to the SUT
- The SUT replies to both Testers, T1 and T2
- The SUT first sends B to T1, then C to T2
- The communication channel adds a delay on B
- Therefore the Test Sequence is {T1!A,T2?C,T1?B}



Why do we need a concurrent test architecture?

- The non-deterministic behaviors of the SUT and the channel delays yield to a set of possible sequences.
  - This trivial example yields to 2 possible outcomes.
- Having this kind of alternatives would soon generate very complex non-concurrent test case descriptions.

```
testcase TC_NonConcurrent_01()
runs on HostType {
  T1.send(A);
  alt {
        [] T1.receive(B){
                  T2.receive(C)
        }
        [] T2.receive(C){
                  T1.receive(B)
        }
      }
  // other events ...
}
```

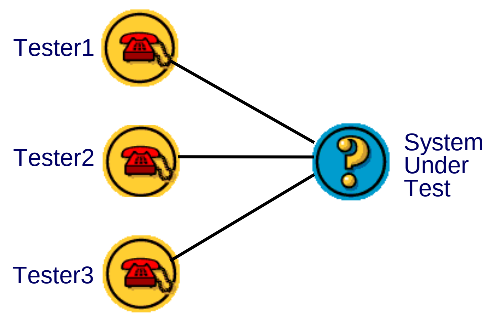

Why do we need a concurrent test architecture?

- Conformance testing:
  - A PBX must accept 12 simultaneous connection requests.
  - A railroad switching controller must compute inputs from 4 detection devices and give feedback.
- Service, function and feature testing:
  - Establish a 3-way conference.
- Stress, robustness and load testing:
  - System must accept 13 simultaneous Service Requests multiple times during a sustaining period of time.

TTCN-3

What kind of architecture can be used?

- Architecture with multiple testers of the same type with only one interface.

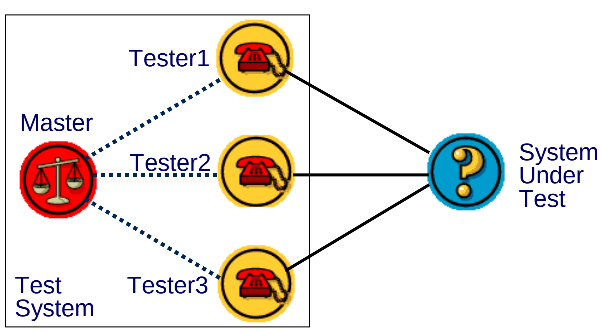


Telelogic

TTCN-3

What kind of architecture can be used?

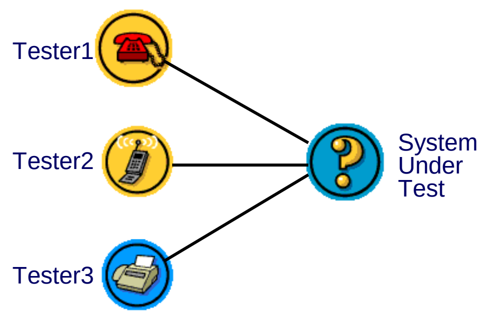
- All testers used the same set of messages and interfaces: one port definition.
- All testers are identical: one component type.



Telelogic

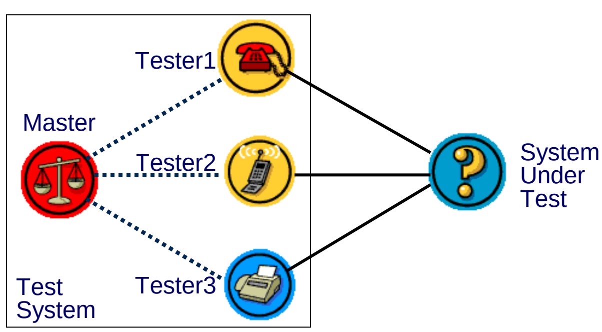
What kind of architecture can be used?

- Architecture with multiple testers of different type.
- Each tester uses its own unique interface.



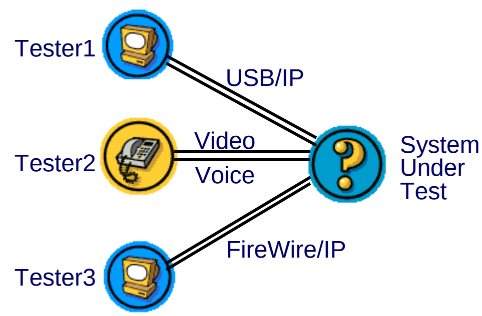
What kind of architecture can be used?

- Each tester uses different set of messages and interfaces: multiple port types.
- Each tester is different: multiple component types.
- But one port type per component type.



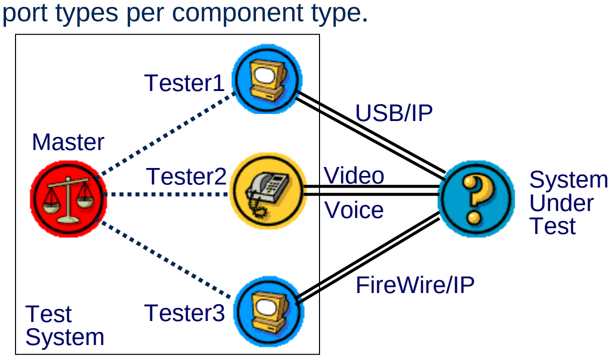
What kind of architecture can be used?

- Architecture with multiple testers of different types.
- Each tester type can have multiple kind of interfaces.



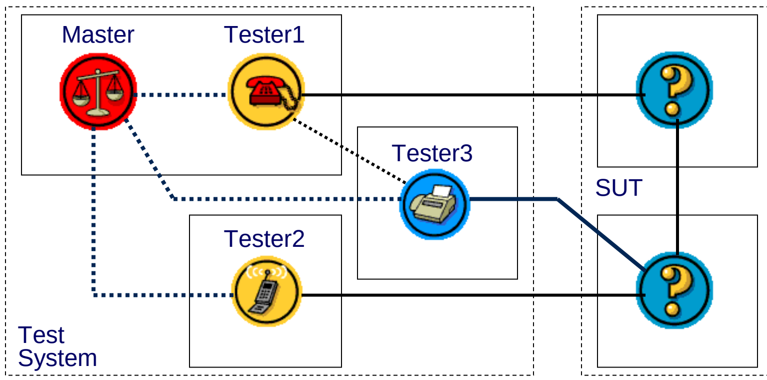
What kind of architecture can be used?

- Each tester uses different set of messages and interfaces: multiple port definitions.
- Each tester is different: multiple component types.
- Multiple port types per component type.



What kind of architecture can be used?

- The Executable Test Suite can be:
  - One Node - Multi-threaded (Simplest, Default)
  - Multi-Node
  - Mixed



How TTCN-3 support such architectures?

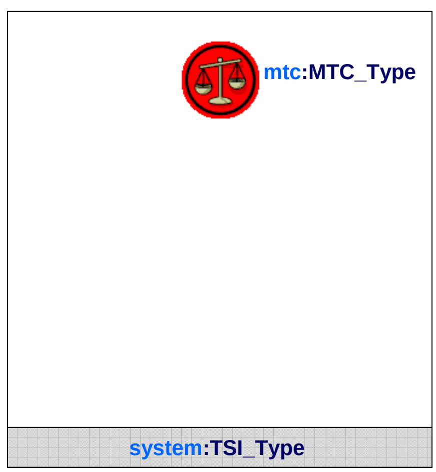
- Dynamic creation of the test configuration
  - Creation of components
    - create
  - Creation of connections between Components
    - map, unmap
  - Creation of connections with the TSI/SUT
    - connect, disconnect
- Dynamic control of the component behavior
  - Control of component behavior
    - start, stop, kill
  - Lookup of component behavior
    - running, done, alive, killed

How TTCN-3 support such architectures?

- Communication between components
  - Exchange of messages between components
    - send, receive
  - Implicit verdict mechanism
    - setverdict, getverdict
    - none, pass, inconc, fail, error

A TTCN-3 Example

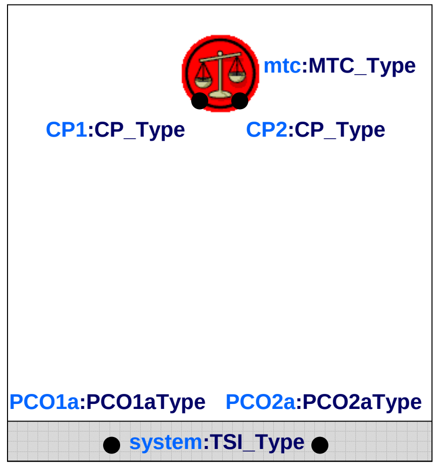
```
// Behavior description
testcase TC_Concurrent_01()
runs on MTC_Type
system TSI_Type {
  ...
}
```



A TTCN-3 Example

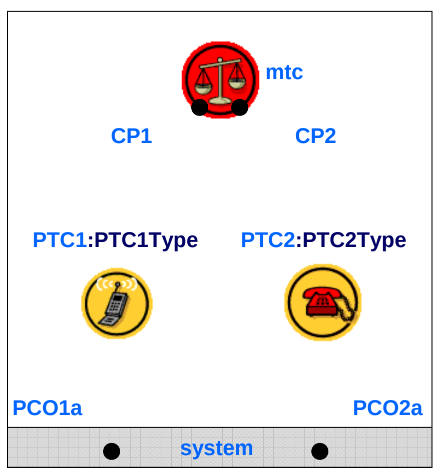
```
// Behavior description
testcase TC_Concurrent_01()
runs on MTC_Type
system TSI_Type {
  ...
}
```

```
type component MTC_Type {
 port CP_Type CP1;
 port CP_Type CP2;
}
type component TSI_Type {
 port PCO1aType PCO1a;
 port PCO2aType PCO2a;
}
// other components ...
type port CP_Type message {
 inout // messages ..
}
// other ports ...
```



A TTCN-3 Example

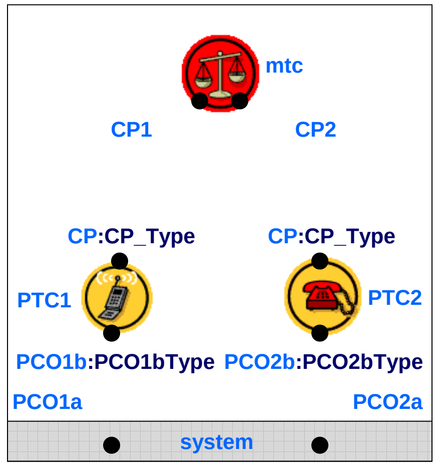
```
// Behavior description
...
PTC1 := PTC1Type.create
PTC2 := PTC2Type.create
...
```



A TTCN-3 Example

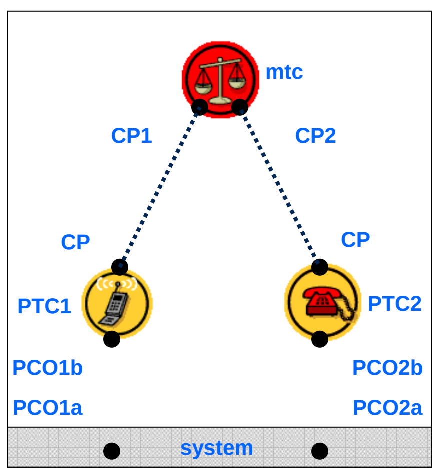
```
// Behavior description
...
PTC1 := PTC1Type.create
PTC2 := PTC2Type.create
...
```

```
type component PTC1Type {
 port CP_Type CP;
 port PCO1bType PCO1b;
}
type component PTC2Type {
 port CP_Type CP;
 port PCO2bType PCO2b;
}
// other components ...
type port PCO1bType message {
 inout // messages ..
}
// other ports ...
```



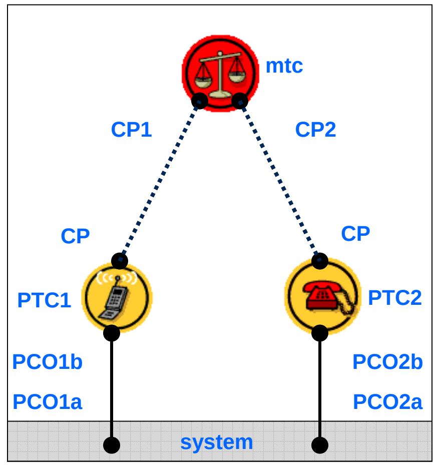
A TTCN-3 Example

```
// Behavior description
...
connect(mtc:CP1, PTC1:CP);
connect(mtc:CP2, PTC2:CP);
...
```



A TTCN-3 Example

```
// Behavior description
...
map(PTC1:PCO1b, system:PCO1a);
map(PTC2:PCO2b, system:PCO2a);
...
```

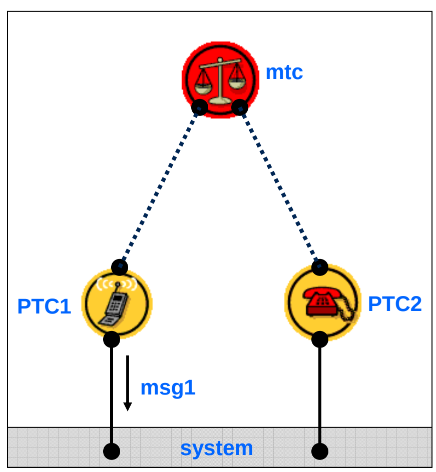


A TTCN-3 Example

```
// Behavior description
...
PTC1.start(TS_InitiateCall());
PTC2.start(TS_AnswerCall());
...
```

```
function TS_InitiateCall()
runs on PTC1Type {
  ...
  PCO1b.send(msg1);
  ...
}
```

```
function TS_AnswerCall()
runs on PTC2Type {
  ...
  PCO2b.receive(msg2);
  CP.send(statusConnected);
  ...
}
```

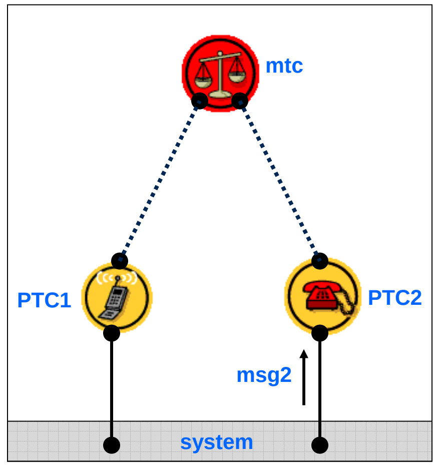


A TTCN-3 Example

```
// Behavior description
...
PTC1.start(TS_InitiateCall());
PTC2.start(TS_AnswerCall());
...
```

```
function TS_InitiateCall()
runs on PTC1Type {
  ...
  PCO1b.send(msg1);
  ...
}
```

```
function TS_AnswerCall()
runs on PTC2Type {
  ...
  PCO2b.receive(msg2);
  CP.send(statusConnected);
  ...
}
```

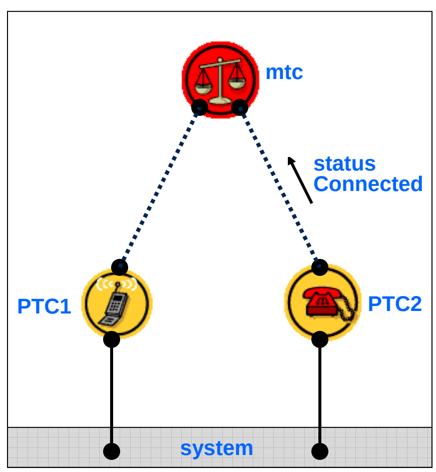


A TTCN-3 Example

```
// Behavior description
...
PTC1.start(TS_InitiateCall());
PTC2.start(TS_AnswerCall());
...
```

```
function TS_InitiateCall()
runs on PTC1Type {
  ...
  PCO1b.send(msg1);
  ...
}
```

```
function TS_AnswerCall()
runs on PTC2Type {
  ...
  PCO2b.receive(msg2);
  CP.send(statusConnected);
  ...
}
```



TTCN-3

Creating normal component

- Components are automatically destroyed at the end of the executed behavior function or when stopped

```
var PTCType ptcname;
ptcname := PTCType.create("InstanceName");
... // connect, map, ...
ptcname.stop;
ptcname := PTCType.create("InstanceName");
... // connect, map, ...
ptcname.done;
ptcname := PTCType.create("InstanceName");
... // connect, map, ...
```


TTCN-3

Creating alive-type component

- Alive Components can execute multiple behavior functions
- Components are not destroyed when stopped or when there behavior is done

```
var PTCType ptcname;
ptcname := PTCType.create("InstanceName") alive;
... // connect, map, ...
ptcname.start(TS_BehaviorTwo());
ptcname.done;
ptcname.start(TS_BehaviorThree());
ptcname.done;
ptcname.kill;
ptcname := PTCType.create("InstanceName") alive;
... // connect, map, ...
```

Connecting and mapping

- After creation of the components we need to **connect** ports between MTC/PTC components and **map** ports between an MTC/PTC component and the Test System Interface – TSI
  - The **mtc**-keyword identifies the MTC, **system** identifies the TSI instance and the **self**-keyword identifies the currently executing MTC/PTC
- Without connecting/mapping a component cannot communicate with the outside world
- When **connecting** port A and port B, the **in** list of port A must match the **out** list of port B and vice versa
- When **mapping** port A and port B, the **in** list of port A must match the **in** list of port B, and the **out** list of port A must match the **out** list of port B

Unconnect and Unmap

- Connections and Mappings can be undone, to change configuration during the runtime of the test
- Syntax is the same as for connect and map

Starting and Stopping test components

- Once components are created and connected/mapped, they can be started
- The behavior to be executed by the component is given in the **start** command
  - The behavior is defined as a function
- Components can be stopped using the **stop** command
  - Only the execution of test behavior is stopped.
  - Components can stop themselves, or other components
- Components can be destroyed using the **kill** command
  - The execution of test behavior is stopped - if any
  - All associated resources (including all port connections) are freed
  - Components can kill themselves, or other components

Querying test components

- The **running** operation returns a boolean value based on whether the component is running or not
- The **alive** operation returns a boolean value based on weather the component is already executing or ready to execute behavior, or not
- The **done** operation can only be executed when the component has completed its behavior
- The **killed** operation can only be executed when the component has been destroyed

Details from ETSI ES 201 873-1 v3.2.1

Table 15: Overview of TTCN-3 configuration operations

Operation	Explanation	Syntax Examples
Connection Operations		
connect	Connects the port of one test component to the port of another test component	`connect(ptc1:p1, ptc2:p2);`
disconnect	Disconnects two or more connected ports	`disconnect(ptc1:p1, ptc2:p2);`
map	Maps the port of one test component to the port of the test system interface	`map(ptc1:q, system:sutPort1);`
unmap	Unmaps two or more mapped ports	`unmap(ptc1:q, system:sutPort1);`
Test Component Operations		

create	Creation of a normal or alive test component, the distinction between normal and alive test components is made during creation (MTC behaves as a normal test component)	Non-alive test components:
		`var PTCType c := PTCType.create;`
		Alive test components:
		`var PTCType c := PTCType.create alive;`

start	Starting test behaviour on a test component, starting a behaviour does not affect the status of component variables, timers or ports	`c.start(PTCBehaviour());`
stop	Stopping test behaviour on a test component	`c.stop;`
kill	Causes a test component to cease to exist	`c.kill;`
alive	Returns true if the test component has been created and is ready to execute or is executing already a behaviour; otherwise returns false	`if (c.alive) …`
running	Returns true as long as the test component is executing a behaviour; otherwise returns false	`if (c.running) …`

Details from ETSI ES 201 873-1 v3.2.1

Operation	Explanation	Syntax Examples
done	Checks whether the function running on a test component has terminated	`c.done;`
killed	Checks whether a test component has ceased to exist	`c.killed { … }`
Reference Operations		
mtc	Gets the reference to the MTC	`connect(mtc:p, ptc:p);`
system	Gets the reference to the test system interface	`map(c:p, system:sutPort);`
self	Gets the reference to the test component that executes this operation	`self.stop;`

Tips and Guidelines

- Common behavior must be defined in function

```
function TS_SetupConnection()
runs on PTC1Type {
  ...
  PCO1.send(msg1);
  ...
```

- Theses functions can be called by any other function running on the same component type.
- These function should be parameterized with the PCO and CP that they use.

```
function TS_SetupConnection(pco:PCOType)
runs on PTC1Type {
  ...
  pco.send(msg1);
  ...
```


Tips and Guidelines

- It is strongly recommended to check that the PTCs have finished their execution, with the use of the DONE statement in MTC, before terminating the MTC.

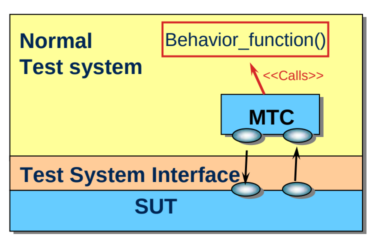
```
all component.done;
setverdict(pass);
stop;
```

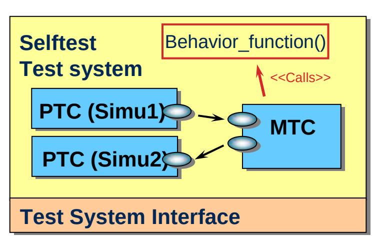
Tips and Guidelines

- There is no need to explicitly passed PTC verdicts to the MTC using coordination messages
  - A global verdict is automatically maintained by the MTC
  - The global verdict is updated whenever a component terminates
  - Remember: Verdict never improve
  - Make the TTCN-3 script more readable

Testing Concept: Self-test of Test Cases

- Use Concurrency to perform a self-test of a test case
  - All behavior is encapsulated in a function. In the normal case, this function is simply called in the MTC
  - For Self-Testing, a Simulation of each of the SUT Ports is implemented in one or more Parallel Test Components (PTCs). They are connected to the MTC ports
  - Since the Test System Interface can be left empty, SUT Adaptation is not needed for the self-test test suite





Benefits with Concurrent TTCN

- Less code to write
- Can have several test architectures in the same test suite
- Several service providers can be used
- Other components can be created at any time during the test case execution
- Concurrency
  - We can have several components executing simultaneously
  - Several processes aiming at the same goal