Network Element Testing Using TTCN-3: Benefits and Comparison

G. Bhaskar Rao[1], Keerthi Timmaraju[1], and Thomas Weigert[2]

[1] Motorola India Electronics Ltd
Hyderabad, India
{bhaskarraog, keerthi.t}@motorola.com
[2] Motorola Inc.
Schaumburg, Illinois
thomas.weigert@motorola.com

Abstract. As testing often consumes over 40% of the typical project development effort, there is great need for optimizing the testing effort. In addition, as the cost of fixing defects is dramatically lower when fixing those close to where they were introduced, finding defects in the early life-cycle phases is critical. TTCN-3 (Testing and Test Control Notation), developed at ETSI and standardized by the ITU-T, enables testers to specify test cases for the various types of testing, and supports reuse of test artifacts. We have used TTCN-3 as a complete test solution in the development of network element software. This paper presents the benefits we have observed during system development and provides a comparison with other testing practices deployed in our organization.

1 Introduction

Testing consumes typically over 40% of the total software engineering effort in telecommunication system development. A typical breakdown of the total test effort is shown in Table 1, as based on our experience in developing telecommunication systems (the data in this table represents our base line of expected effort as averaged from a reasonably large number of similar development projects). These development projects traditionally have used languages such as C, Perl, or Tcl to specify test suites and implement test environments. From this table, it is apparent that most of the effort is spent on developing the environment for carrying out the overall test activity, followed by the development of the test cases. It is also clear that most of the effort is spent in activities other than testing of the system under test. Test teams typically use or develop different tools and environments for integration testing, performance testing, conformance testing, and load testing, with minimal reuse between them, or no-reuse at all. When a defect is detected, it takes considerable time to associate this defect with the appropriate aspect of the system under test due to the hand-crafted test environment, difference in environments, different test scripts, and the manual effort of tracing tests to requirements, design, or code artifacts. It is plain that development projects could save significant efforts were they to spend time only on test objects by using standard test environments, which have the capability to support different types of testing activities, rather than developing custom environments every time again.

The results of an earlier pilot project in protocol implementation encouraged us to rely on TTCN-3 and supporting tools for the development of a major release of a telecommunication system. This system required, of course, unit testing, and integration testing. As the developed network elements were performance critical, we also needed to perform rigorous performance testing, conformance testing, load testing, and reliability testing. We wanted to rely on a single environment that could support these testing needs in a transparent manner and would allow us to reuse as much of the test artifacts as possible. In addition, some of the network elements were developed using a new development methodology (UML 2.0 and supporting tools), and thus, testing also involved profiling the system under test to obtain performance measures such as message queuing times, message processing times, timer delays, etc., under different call load scenarios, in order to obtain insights about the adequacy of this methodology.

Table 1. Effort Distribution in conventional testing

Test Activity	**Effort Spent**
Test architecture	7%
Test design	10%
Test case identification	8%
Test case development	20%
Cost of Quality of Test System	7%
Communication, encoders and decoders	8%
Test Environment (Logging, Tracing, Defect detection support, Validation, Regression testing, other support activities)	25%
Test Management (Test case Organization, description, communication with customer, etc.)	7%
Other (Learning, procurement, setup)	8%
	100%

This paper summarizes our experiences and the benefits observed of leveraging TTCN-3 in this development project. Section 2 highlights the features of the TTCN-3 language and the supporting tools we deployed. This section also overviews our testing approach and our test architecture. The sample test case in section 3 illustrates various features of TTCN-3. Section 4 describes our development project. In section 5, we give a comparison of the traditional test development approach relying on programming languages such as C, Perl, or Tcl with the TTCN-3 approach. We conclude with a summary of the impact TTCN-3 had on our testing efforts.

2 TTCN-3

Recent efforts at ETSI have led to the introduction of a common general purpose testing language for the industry: TTCN-3 (Testing and Test Control Notation). While its precursor TTCN-2 was mainly used for communication and network system or subsytem testing, TTCN-3 has a rich set of features which make it suitable for other domains also, such as automotive or telematics applications [5], as well as for different types of testing activities. We believe that TTCN-3 addresses most of the issues raised in section 1.

2.1 TTCN-3 as a Test Solution

The following features of the TTCN-3 language make it suitable for the testing of communication and network systems as well as for other domains.

- Synchronous and asynchronous communication mechanisms help in testing of procedure based and message based systems.
- Data and signature templates with corresponding matching mechanisms provide flexibility to the user to reuse these templates across various test cases.
- The user is able to specify the expected messages with all applicable message parameters required to determine that a test case has passed.
- Separation of test case specification from execution control. The same test case can be executed in a loop at specific time intervals, or it can be grouped with other test cases, or it may be sequenced for stress testing, and so on. Each test case can thus be independently controlled.
- Dynamic concurrent testing configurations provide the user with a flexible option to simulate the behavior of unavailable components (for example, components that are still under development). This feature also helps in writing the test case in a more realistic scenario in the presence of concurrent components.
- Encoding information can be specified along with the test case. Note that at times the same message has to be encoded or decoded differently, depending on the context.
- Test cases may be written in programming languages (such as C), MSC notation, or the tabular format familiar from TTCN-2.
- External code integration provides the flexibility to integrate legacy encoders or decoders, code libraries, transformations, etc.
- Regular expressions greatly simplify the specification of expected messages
- Extensions to implement automatic configuration of the system under test (SUT) using SUT operations.
- Finally, TTCN-3 is a standardized language supported by commercial tools, such as Telelogic Tau Tester or Testing Technologies TTThree.

2.2 Overview of Tau Tester

Telelogic Tau/Tester is a tool for designing, creating, and executing TTCN-3 test suites. It includes editors for TTCN-3, ASN.1, and text. The tool provides build facilities, an integrated MSC viewer, and a log file creator. The major features of Tau/Tester are as follows.

- Support for TTCN-3.
- Support for ASN.1 PER (aligned and unaligned) and BER (definite and in-definite) encoding and decoding rules.
- TTCN-3 encoding/decoding must be written manually.
- Integrated development environment.
- On-line help.
- C code Generator.
- Support for TRI and PL (proprietary) integration mechanisms.
- Provides logging, document generation, and recording of the test execution.

2.3 Test Environment Architecture

The typical test system architecture and components/tools involved in testing are shown in fig. 1. TTCN-3 files which comprise the test system architecture, its behavior, data and control, along with the adaptation code (encoder/decoders and communication between test system and SUT), are processed by the TTCN-3 tool which generates code, produces a makefile, and compiles the test system. Test cases can be controlled by the user through the execution control UI. The communication between test system and SUT can be implemented using standard TCP/UDP communication links or proprietary protocols.

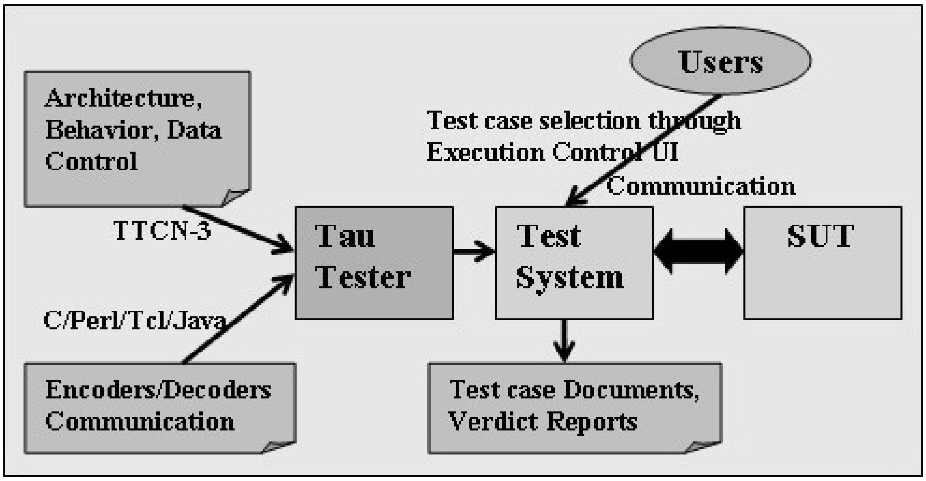

Users
Architecture, Behavior, Data Control
Test case selection through Execution Control UI
Communication
TTCN-3
Tau Tester
Test System
SUT
C/Perl/Tcl/Java
Encoders/Decoders Communication
Test case Documents, Verdict Reports

Fig. 1. Test Environment Architecture

2.4 Test System Architecture

Figure 2 shows the components of the executable Test System. TTCN-3 generated code executes on top of the runtime system libraries that implement the abstract constructs of the language. The runtime system controls the execution, it encodes/decodes messages using appropriate codecs, and logs system events via the log management system. Communication with the SUT is through the communication system.

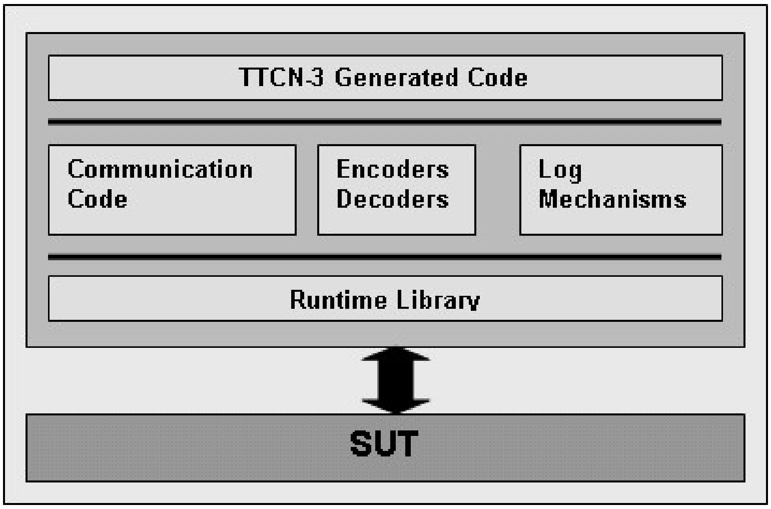


Fig. 2. Test System Architecture

2.5 Test Development Process

Figure 3 outlines the major phases of test development. Sequence Diagrams or MSC/HMSC [6] are often used during the test requirements phase. Using these notations, both valid and invalid test scenarios can be described easily. During the architecture phase, a choice has to be made between multi-threading (the multi component/concurrent model) and the simple/single component model. In general, for integration or system testing, a single component model meets most of the requirements. The concurrent model, on the other hand, is well-suited for load testing. In a test architecture following the concurrent model, the test verdict depends on the verdicts for the individual components. The test data is represented using templates and passed as arguments to the messages. Parameterized templates and regular expressions may help to increase the reuse of test data and test cases. The test cases can be called in sequence to form an integration suite.

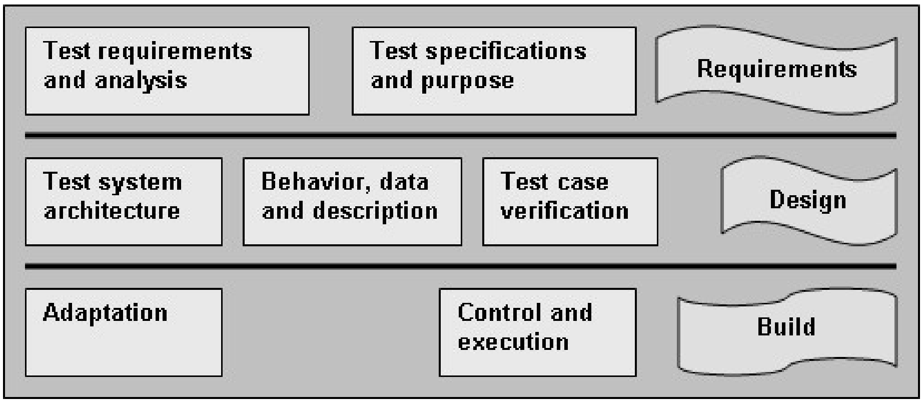


Fig. 3. Phases in test system development

3 Sample Test Case

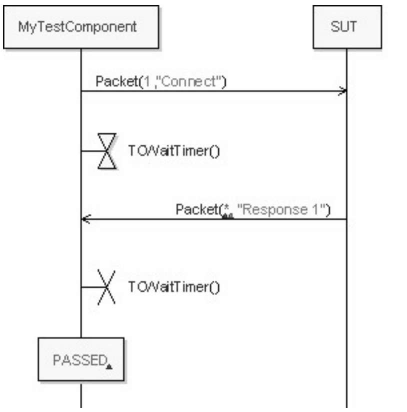
From the example test case [1] below we can easily see the the various aspects of a TTCN test: architecture, behavior, and control. The detailed test description can be seen from the sequence diagram and objectives table in fig. 4.

```
module sampleTC_valid
{
  // Data Definitions
  type record   Packet
  {
        integer   info,
        charstring data
  }
  type port  DataPort message
  {
     inout all;
  }
  // Test Component declaration PTC
  type component MyTestComponent
  {
     port DataPort  CompPort;
     timer TCWaitTimer:= 100.0; //seconds
  }
  type component SystemComponent
  {
     port DataPort SysPort ;
  }
  // template definitions
  template Packet send_message :=
  {
     info := 1,
     data := "Connect"
  }
  template Packet expected_message :=
  {
     info := *,
     data := "Response 1"
  }
  /*  test case TC_01 */
  testcase TC_01() runs on MyTestComponent // defines MTC
  system SystemComponent
  {
     log("Start test case execution for TC_01");
     map(mtc: CompPort, system: SysPort);
     CompPort.send(send_message);
     TCWaitTimer.start;
     alt
     {
            [] CompPort.receive(expected_message){
                  TCWaitTimer.stop;
                  setverdict(pass)
        }
            [] any port.receive{
                  TCWaitTimer.stop;
                  setverdict(fail)
        }
            [] MaxTimer.timeout{
                  setverdict(fail)
        }
     }
     unmap(mtc: CompPort, system: SysPort);
  }
  control  /* control part of the module */
  {
     execute (TC_01 ());
  }
} /* end of the module */
```

The architecture part of this module begins by declaring a simple message data structure referred to as a packet, comprised of an `info` integer field followed by the `data` characterstring. Then a simple port type able to convey arbitrary bidirectional messages is declared. We then describe two components, the test driver and the component representing the SUT. These communicate via the identified ports.

Then two templates for messages are defined: A message to be sent to the SUT, and the expected reply message. The latter defines the pattern a message received from the SUT has to satisfy to be recognized as a reply. In this particular case, a reply message may have an arbitrary info field, as indicated by the `*` (wildcard) symbol, but must have `Response 1` as data. In the control behavior section, a simple test case is defined. Upon invocation, the connection between test driver and SUT is established, the first message above is sent to the SUT, and a timer is started. The test component now is waiting for one of three events: Either the defined reply message is received at the appropriate port, upon which the timer is stopped and the test case is considered to have passed. If any other message is received on any port, or the reply message is received on any other port, the timer is stopped and the test case is considered to have failed. Similarly, if the timer expires without the reply message having been received, the test case is considered to have failed. Then the connection between test component and SUT is deleted.

Finally, the control part simply tells us to execute that single test case.



Identifier	MSC_TC_Valid
Reference to requirement	Requirement #1
Initial Condition	Initially in Idle State
Checks to be performed	Test component should now address a port of SUT.
Verdict criteria	Message has been received successfully

Fig. 4. Sample MSC and Objectives Chart

4 Case Study Overview

This paper presents a case study of testing a basic network element developed using UML 2.0 for a high-availability target platform [3]. It also outlines the benefits of TTCN-3 as compared with conventional testing practices using languages such as C, Perl, or Tcl.

In this project, we developed a new mobility management layer for a CDMA network, with high availability and scalability to meet next generation demands. The project involved development of the call processing stack, as well as mobility management, resource management, and link management components of the core network.

The call processing layer was architected using configurable working threads to share the call load (see fig. 5). This layer was developed from UML and implemented on a High Availability platform. A main concern of this implementation was the ability to handle are large call load, and be flexible to support further increasing call loads. Calls are routed by the main thread (Router Thread) to call processing threads (labelled Thr1, Thr2, etc., in fig. 5), which then process these calls with the help of supporting threads. The Router Thread performs load-balancing across the call processing threads. The number of call processing threads can be configured dynamically depending on the call load.

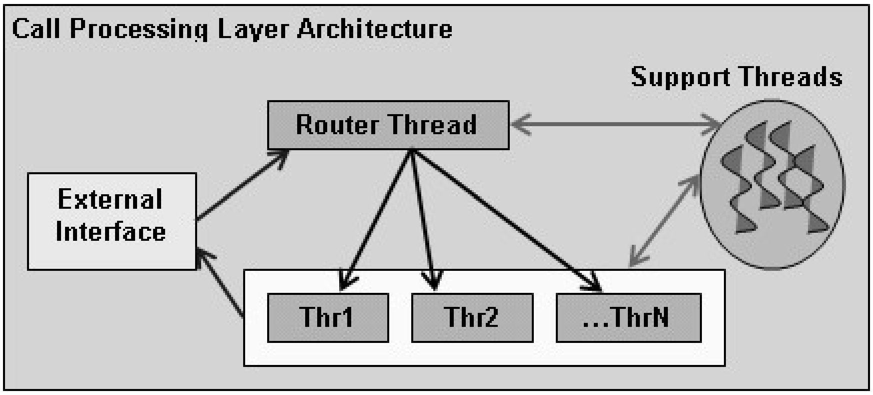


Fig. 5. Architecture of the System Under Test

Test cases were developed for integration testing, system testing, component testing, and load testing. The load system used two components to simulate two interfaces of the system. Along with the test system, user defined library functions had to be integrated to calculate the response times of the SUT. The integration test cases were also used for system testing by systematically integrating each module and interface.

4.1 Architecture

The Abstract Test System may have either one or two components in addition to the encoder/decoder (Adaptation Layer) with ports for message exchange with the SUT (see fig. 6). The Abstract Test System Interface (ATSI) receives two kinds of messages; hence there are two ports, one for each kind of message. The test system ports establish a TCP connection with SUT ports or use UDP data packets to exchange messages with the SUT.

The static test execution setup is shown in fig. 7. It shows the system under test (right-hand side) and the Test System (left-hand side) communicating via a TCP/IP connection. The SUT is comprised of application code (in this case generated from UML 2.0 designs), encoder/decoder, tool-specific run-time library,

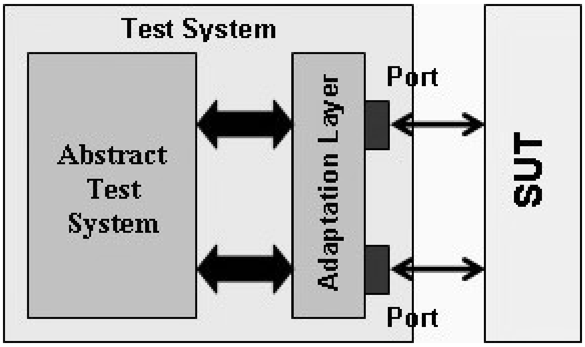


Fig. 6. Test System Architecture

and a communication module, whereas the major components of the test system are the TTCN-3 generated code, the TTCN-3 run-time library, encoder/decoder and the communication module. In-coming messages are sent to the application layer after decoding by the respective decoders. Out-going messages are encoded by the respective encoders and then sent to the target system. As there are two types of messages being exchanged, both systems have two threads for receiving each type of message. The Telelogic Runtime Library simplified the creation of these threads by providing appropriate hooks, for both SUT and the test system. The generated application code too executes on separate threads; the underlying Runtime Library provides mutual exclusion for all these interacting threads.

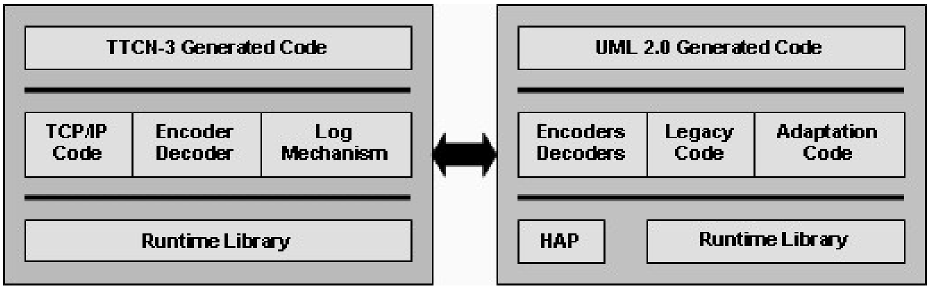


Fig. 7. Static Test Execution Setup

Both single (S-TTCN) and multi component models (C-TTCN) are used for load testing; the single component model is used for integration and system testing. During load testing, the SUT receives messages from different test systems, instead of a single test system as in conformance testing and integration testing. The dynamic execution setup for load testing is shown in fig. 8.

A TTCN-3 implementation has 4 modules: Data types, architecture, behavior, and control modules. The external functions are defined in a separate module. The relationships among the modules are shown in fig. 9.

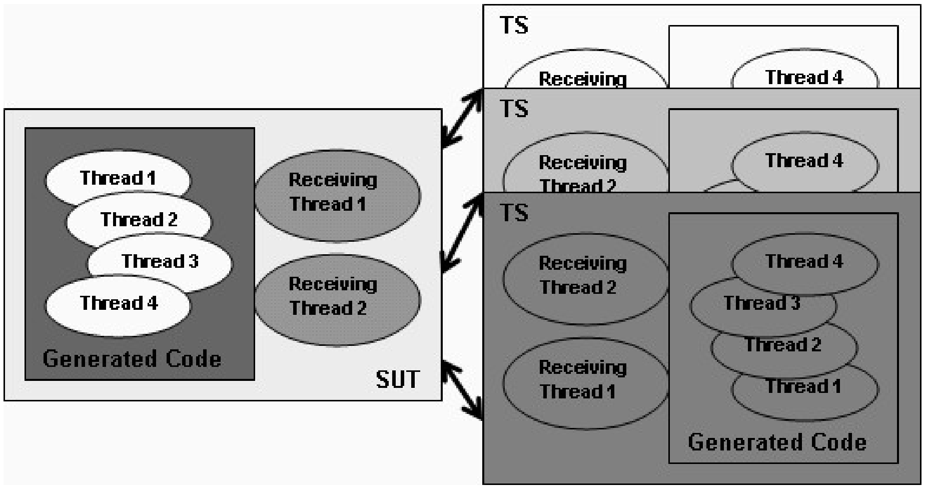


Fig. 8. Load testing execution setup

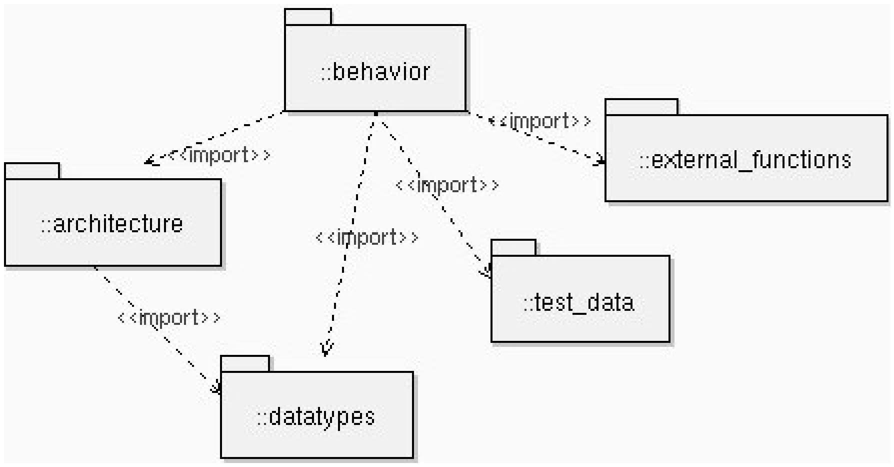


Fig. 9. Relationships among TTCN-3 modules

4.2 Test Case Development

For integration testing, separate test cases were developed for call setup, call termination, and hand-off. Both success and failure test cases were defined to gain further confidence in the system behavior. Most of the system test cases were obtained by reordering and combining the individual integration test cases. For example, the reference MSC in fig. 10 below shows that the system test case “end-to-end Call Test” results from the integration of the three basic test cases for call setup, handoff, and call termination. Such integration is achieved in

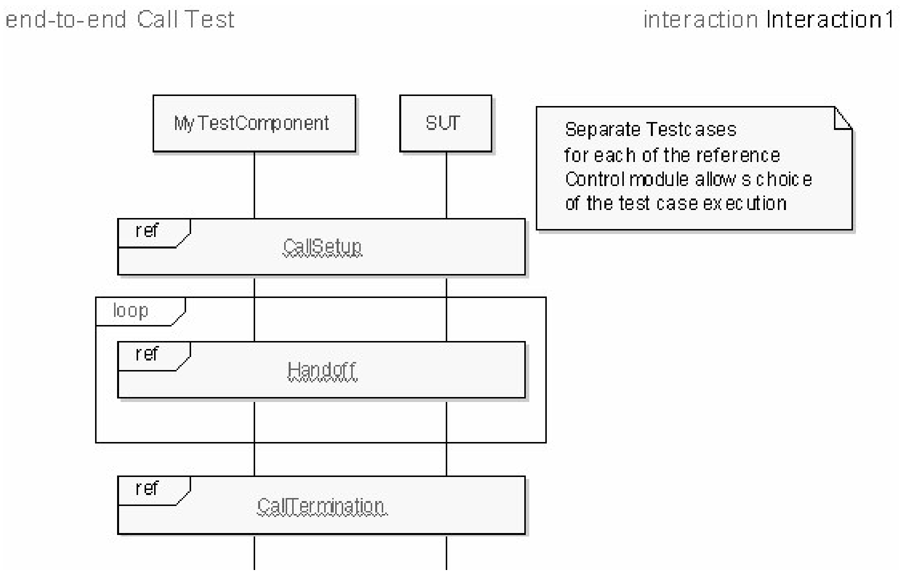


Fig. 10. Reference MSC for Call Setup

the control module; in addition, some templates were changed to obtain invalid behaviors to ensure better coverage of the SUT.

4.3 Load Testing

After ensuring that the system is initialized properly, the test system generates the first call setup message following the single component load test system model. Calls are generated as per the configuration at different rates (from one call per second to 30 calls per second) with other messages interleaved. Timers are used to configure the load on the system. Since the structure of each message was similar, a message type was created, and all messages contained this structure as their parameter. The contents of this structure was changed whenever messages were exchanged by the corresponding systems. On the concurrent component model, the main test component creates the call generator component after every interval, which in turn generates one call and dies after termination of that call. External functions were used to measure the time taken before sending and after receiving messages. These functions measure the performance of the SUT as well as of the test system. Based on performance measurements, the call rate was increased or decreased. The result of each test case was logged to a file.

5 Comparison with Conventional Testing

Languages such as C, Perl, or Tcl are not primarily intended as testing languages, but they do enable us to write test cases. Often engineers think that testing is merely calling a function to send a message and later comparing the result obtained with the expected result; this mentality eventually leads to ad-hoc testing. In such testing, different test environments are often repeatedly developed, and unnecessary logic for comparisons often reduces the time available for implementing test cases and the test system. Conventional languages provide few or no supporting facilities to go beyond ad-hoc testing and to manage testing in a systematic way. Developers have to visualize, plan, and implement everything starting with architecture, the separation of encoding/decoding from behavior, communication, comparison logic, logging, data management, reporting, document generation and so on. However, most of these features are provided directly by TTCN-3, which may immediately impact various business parameters (see Table 2).

Table 2. TTCN-3 impact compared to that of conventional languages

Business Parameters	**Conventional Testing**	**TTCN-3**
Productivity	1x	2x (Better)
Impact on Quality	1x	2x (Better)
Impact on CTR	1x	1.5x (Better)
Reuse	1x	2x (Better)
SUT coverage (same effort)	60%	90%

These parameters were estimated before the project and have been verified by other projects. Test coverage was estimated to be at 90% with the same amount of test effort, based on the baselines of the organization (as compared to 60% test coverage with conventional testing).

With respect to features of the TTCN-3 language, the following observations surfaced:

– Templates and timer handling enabled good solutions for integration testing, reliability testing, performance testing, and load testing.
– Control logic, modified templates, and concurrency allowed us to write load generation and processing logic conveniently as part of the test case.
– TTCN-3 code is independent of the platform it is developed on, and it further is very portable. The same TTCN-3 code was used with another tool, with only minor modifications to integration code (adaptation layer).
– Considerable amount of reuse across different types of testing was achieved, in particular resulting from reusing test cases and templates.
– The cost of quality of the test system was substantially less by virtue of concentrating only on test objectives.
– Generated test systems can be used as back ends because of their easy integration with other system components, developed in arbitrary languages.

Table 3 shows the distribution of the total test effort for projects leveraging TTCN-3. From this we can conclude that within a given time one can develop more test cases with better quality using TTCN-3, as compared with the conventional approach (in Table 1). While a new network element was developed during this case study, we feel that the data observed is representative of telecommunication system software in general.

In our experience, the test effort spent on projects following the conventional approach is roughly 1.5 times the effort spent in projects developing test suites using TTCN-3 and leveraging a TTCN-3 execution environment.

Table 3. Effort Distribution using TTCN-3

Test Activity	**Conventional**	**TTCN-3**
Test architecture	7%	8%
Test design	10%	7%
Test case identification	8%	15%
Test case development	20%	45%
Cost of Quality of Test System	7%	7%
Communication, encoders and decoders	8%	8%
Test Environment (Logging, Tracing, Defect detection support, Validation, Regression testing, other support activities)	25%	-
Test Management (Test case Organization, description, communication to customer etc.)	7%	5%
Other (Learning, procurement, set-up)	8%	5%
	100%	100%

Table 4 further summarizes the impact of TTCN-3 on testing based on the test projects done in our organization.

6 Conclusions and Recommendations

TTCN-3 enabled the development of a test environment which supported the various types of testing required and the reuse of test artifacts between these test efforts. Further opportunities for automation were identified and implemented, such as the generation of proprietary encoders/decoders and the generation of TTCN-3 data types from UML 2.0 data types.

Based on our experience and observations from this and similar projects, we feel that test automation with TTCN-3 can be beneficially employed for module testing, integration testing, performance testing, conformance testing, and load testing of communicating and event driven systems. Though TTCN-3 is claimed to be general purpose, some enhancements are required to truly make it suitable for testing GUI and data base systems.

We strongly feel that TTCN-3 is well suited for testing in the infrastructure domain. Not only did it help the testing and development teams to generate test cases faster, but also, debugging of test cases became easy. The benefits are significant for medium and long-term projects (but impact is harder to assess for projects with short cycle times). We expect the emergence of TTCN-3 as a prominent testing technology, across a wide variety of domains.

Table 4. TTCN-3 Impact on testing

Feature	**Aspects**	**Our Rating**
Test architecture	The framework provides good mechanism with Simple TTCN and Concurrent TTCN	Excellent
Test design	The design is modular and independent from the platform	Excellent
Test case specification	No explicit support but MSC can be used extensively to document the test cases	Very good
Test data	Very good support with templates, parameterized templates etc.	Excellent
Test execution	Good support for execution of a test cases from the test control block	Very good. Scope for some improvements
Modifiability of test cases	Templates, modular development	Excellent
Test reporting	Indicates which test have failed and passed, with reasons and byte information	Good.
Reuse	Test case and data level reuse	Excellent
Log management	Supports MSC and text based logging	Excellent
Ease of learning of the language	4 days of learning and practice are needed	Satisfactory
Support for ASN.1	Support for ASN.1 data types	Excellent
Support for Encoder/Decoder generation	Automatic generation of support for encoders/decoders from TTCN-3, ASN.1 and mixed types	Very good for ASN.1
Support for test management	Support dynamic selection of test cases	Good
Support for test case verification and validation	Compilation of the test cases, definitions etc	Very good
Scope for further automation	TTCN-3 encoder/decoder, structures, generation of test cases from requirements, test management integration etc.	Excellent
Legacy and External code Integration in Test System	Library Integration and encoders/decoders for the library parameters have to be written explicitly	Bad

References

1. ETSI ES 201873, The Testing and Test Control Notation TTCN-3, Part 1 (Core Language), Part 2 (Tabular Format), Part 3 (Graphical Format), Part 4 (Operational Semantics), Part 5 (TTCN-3 Runtime Interface), Part 6 (TTCN-3 Control Interfaces) (http://www.etsi.org/)
2. Telelogic Tau Tester: TTCN-3 to C Code Generator, Reference Manuals. Telelogic 2004.
3. Peter Weygant, *Clusters for High Availability: A Primer of HP Solutions*, Second Edition, Hewlett-Packard Company, 2001.

4. Telelogic Tau G2: UML 2.0 to C/C++/Java Code Generator, Reference Manuals. Telelogic 2004.
5. Simon Burton, Andrea Baresel and Ina Schieferdecker, *Automated testing of automotive telematics systems using TTCN-3*, 3rd Workshop on System Testing and Validation - SV04, Paris, December 2004.
6. ITU-T Z.120 (2000): Message Sequence Charts, MSC-2000, Geneva, October 1999.
7. Roger S. Pressman, *Software Engineering A Practitioner's Approach*, Fourth Edition, McGraw-Hill, 1997.
8. Paul Baker, Ekkart Rudolph and Ina Schieferdecker, *Graphical Test Specification - The Graphical Format of TTCN-3*, Tenth International SDL FORUM, Copenhagen, June 2001.
9. ITU-T Recommendation X.680 (1997): "Information Technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation". Geneva, October 2000.
10. OMG: Unified Modeling Language v2.0 (UML). Object Management Group. 2003.