EXPERIENCES OF TTCN–3 TEST EXECUTOR DEVELOPMENT

János Zoltán Szabó
Ericsson Research
Conformance Center, Ericsson Ltd.
H-1037 Budapest, Laborc u. 1., Hungary
Szabo.Janos@eth.ericsson.se

Abstract The new generation of communication protocols requires new test methods. The third version of Tree and Tabular Combined Notation (TTCN–3) embodies such a method. Because of conceptual changes, the existing TTCN–2 test environments cannot be reused; completely new tools shall be built on new bases. This paper shares our experiences gained during the design and usage of a TTCN–3 test executor developed for Ericsson interim use. The system architecture and basic ideas of implementation are described in details. We also present some case studies to demonstrate how efficiently the TTCN–3 language and the test executor can be used in real-life testing projects.

Keywords: TTCN–3, test execution, conformance testing

1. INTRODUCTION

Tree and Tabular Combined Notation version 3 (TTCN–3) [1] is a new test description language standardized recently by ETSI. TTCN–3 is the successor of TTCN–2 [2], which is widely used for conformance testing of telecommunication protocols based on OSI basic reference model. The designers' goals were to extend the new language's application area, simplify its syntax and, of course, provide backward compatibility. TTCN–3 has numerous new features compared to TTCN–2, such as dynamic creation and re-configuration of parallel test components and procedure call based interfaces. The new constructs make the language capable of performance testing in addition to conformance testing. TTCN–3 can be applied not only for testing communication protocols, but distributed systems and APIs as well. The Machine Processable (MP) format of TTCN–2 went under significant changes and became TTCN–3 Core Language (CL) [3]. Instead of textual representation of strictly

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35497-2_31
I. Schieferdecker et al. (eds.), *Testing of Communicating Systems XIV*

defined tables, TTCN–3 CL rather resembles a high-level programming language. Based on the CL several graphical presentations can be defined. Currently there are two standardized presentation formats: the Tabular Presentation Format (TFT), which provides partial compatibility with TTCN–2 and the Graphical Format for TTCN–3 (GFT).

The main goal of our project was to design and develop a software toolset that can execute TTCN–3 Abstract Test Suites (ATSes) on a generic hardware platform (e.g. on a PC or workstation). The work was started in early 2000, in parallel with the development of TTCN–3 language itself. Our prototype development was based on draft standards of TTCN–3 CL published by ETSI. We decided to develop a compiler that translates TTCN–3 Core Language to C++ source code, which has to be extended with a library and some modules written by the user. These pieces of code together form the source code of the Executable Test Suite (ETS).

Our motivations for making a TTCN–3 executor were complex. First of all, in the beginning not only the syntax but the static and operational semantics of TTCN–3 were not settled yet. The implementation of such tool could detect most of the inconsistencies and ambiguities in the language. On the other hand, our goal was to establish an efficient, high performance test execution platform that allows the use of TTCN–3 on its new application areas, including load testing. Finally, the existence of a tool encourages the usage of the language. Thus the new users can gain TTCN–3 knowledge very early by writing, checking and executing many kinds of test suites for various protocols without waiting for commercial tools to appear on the market.

This paper is organized as follows. The next section outlines the system architecture of our TTCN–3 executor. Section 3 describes the mapping of most important TTCN–3 language elements in details and Section 4 presents some case studies, that is, some applications of our test executor in real testing projects. Finally, the conclusion is drawn in Section 5.

2. THE STRUCTURE OF TTCN–3 TEST EXECUTOR

The schematic block diagram of our TTCN–3 test executor can be seen on Figure 1. The test executor itself consists of two major parts: a *TTCN–3 compiler* and a so-called *Base Library*. The third part (*Test Port*) is a C++ program module written by the user. This section presents these blocks in details.

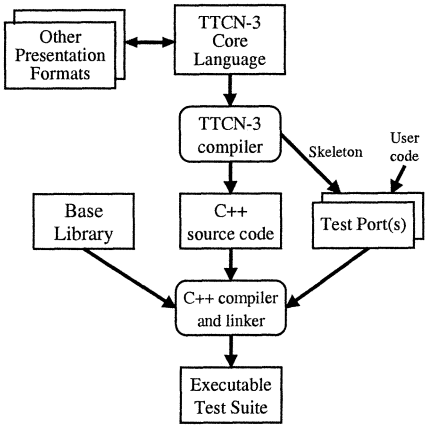

Figure 1. Block structure of the TTCN–3 executor

2.1 TTCN–3 to C++ compiler

The task of the compiler is to transform TTCN–3 CL modules into C++ source code that can be further translated to get an ETS. It has two functional units: The parser part (so-called *front-end*) reads the TTCN–3 input, checks its syntax and breaks it into data structures. The code generator part (*back-end*) synthesizes target (C++) code based on these structures. The parser and the code generator are tightly merged on the level of source code, but their functionality clearly separates.

The front-end is an LALR(1) syntax checker based on the well-proven GNU tools. *Flex* was used for building the tokenizer and *bison* for the grammatical analyzer.

The back-end is written in C language as parser actions. The translation is performed on the fly whenever the right side of a grammar rule becomes available on the top of the parser's stack. The relation between the TTCN–3 language elements and their C++ code equivalents is outlined in Section 3.

The compiler is stateless, which means, when translating a CL element it does not remember the previously passed parts of its input. Thus, the C++ equivalent of a TTCN–3 construct depends only on the original construct and not on any other definitions. The simplicity and the short development time are the most important advantages of this approach. However, the compiler is unable to detect semantic errors in its input and can generate inconsistent target code, which can make the localizing

of TTCN–3 errors extremely difficult. Besides, the TTCN–3 interleaved constructs can not be translated in this stateless manner at all.

2.2 TTCN–3 Base Library

The Base Library holds the common parts of C++ code that is used in all ETSes and hereby simplifies the back-end of the compiler. These basic definitions are written once, but they might be used in all modules of the ETS.

The C++ classes representing timers and built-in TTCN–3 data types, the snapshot handler and logging functions as well as the supporting routines of parallel test execution are implemented in the Base Library.

2.3 Test Ports

The purpose of Test Ports (TPs) is to adapt the test executor to a given protocol, that is, to facilitate the communication between the test executor and the SUT. Test Ports are C++ program modules whose skeleton is generated by the compiler from CL port and type definitions. The user shall write the body of the functions using the primitives of TP Application Programming Interface (API). The TPs are also responsible for the encoding and decoding of PDUs.

Although TPs have similar functionality as SUT Adaptors in the reference model of TTCN–3 Run-time Interface (TRI) [4], they are based on different concepts. Instead of TRI's global approach, the Test Ports perform the adaptation locally – they are coupled with TTCN–3 test components. This eliminates the bottleneck of a centralized API and makes our test executor capable of performance testing as well. Unlike TRI, the adaptors of different protocols are clearly separated in our model, which makes the re-use of existing TPs easier in new test configurations. Based on the socket-related operations of UNIX operating systems, our API provides proper ways for the scheduling of the incoming data processing.

In our API each TP is a class, which implements a TTCN–3 port type. The port instances of TTCN–3 test components are represented by the objects of TP class. All SUT related TTCN–3 port operations – except receiving operations – (start, stop, send, map, etc.) are mapped to member function calls of TP classes. While sending, the outgoing PDU is passed to the appropriate member function as parameter in the internal data representation format of the test executor.

The handling of incoming data is based on an event handler mechanism. Each TP class is provided with an event handler function that can be configured by API primitives to trigger after certain amount of

time or incoming data on given socket file descriptors. When the test execution is blocked waiting for a new snapshot (e.g. because it has reached an alt statement), the snapshot manager of Base Library calls the TP event handler functions whenever a trigger condition is fulfilled. The event handlers can append the decoded PDUs to the port input queue by special primitives.

3. TRANSLATION OF TTCN–3 LANGUAGE ELEMENTS

Our compiler translates each CL element into one or more C++ definitions depending on its kind. The mapping of some TTCN–3 constructs is very simple, but the translation of others is a bit sophisticated. Our translation scheme exploits the object-oriented and the non-object-oriented features of C++.

Each TTCN–3 module is translated into a pair of C++ header and source files and consequently into one object file. In case of modular test suites the resulting object files shall be linked together, that is, the module hierarchy of ETS reflects to the original structure of ATS. In order to support forward referencing, that is, the usage of a definition that is given later – which is allowed in TTCN–3 but not in C++ – the target header and source files are split up into sections. For example, type definitions, function prototypes, global variables are placed into different sections. The number and ordering of sections are fixed, but the ordering of definitions within a section does not matter.

Data types. Both pre-defined and compound TTCN–3 data types are mapped to C++ classes. The classes for predefined types are implemented in the Base Library, while the equivalent classes for user-defined compound types are generated by the compiler using the classes of field types. The C++ equivalents of TTCN–3 constants and variables are objects of their respective type. The built-in TTCN–3 data operations are implemented using the operator overloading feature of the C++ language, and therefore the translated TTCN–3 expressions look very similar to the original. We had to use the same field accessing mechanism in classes representing record, set and union types because the stateless compiler is unable to distinguish them in the source code.

Templates. The templates of each TTCN–3 type are represented by a C++ class that can carry all possible patterns of that type. There is a one-by-one relation between the value classes and the template representation classes (TRCs): the TRCs of built-in TTCN–3 types can be found in the Base Library while the TRCs of compound types are

generated by the compiler based on the TRCs of field types. Each TRC has a type field, which indicates what pattern construct (specific value, complemented list, any value, etc.) is selected. The classes are equipped with member functions that can set the appropriate pattern type and value. The match and valueof operations (which are used in sending and receiving port operations as well) are also implemented in member functions of TRCs.

Our experiments showed that the most frequently used template constructs are the specific values, the any value (?) and the any or omit (*) symbols. That is why the TRCs were designed to be optimal in terms of speed and memory usage with these conditions.

TTCN–3 template instances are mapped to the objects of TRCs. Simple templates become static global objects while parameterized ones are mapped to C++ functions returning an instance of the appropriate TRC. Therefore the formal and actual template parameter lists in C++ resemble their TTCN–3 source. The modified templates are produced by copying and overwriting the ancestor.

Special constructs. TTCN–3 timers are implemented as a C++ class in Base Library. The time is measured internally by UNIX system calls, without user libraries such as Platform Adapters of TRI [4]. The compiler generates the queue handler functions of ports, but the handling of port connections is the same for all port types and therefore it is implemented in the Base Library.

Behaviour definitions. TTCN–3 behaviour definitions, such as functions, test cases, control parts are mapped to C++ functions. The mapping of control constructs, variable instantiations and assignments is almost trivial. Thanks the OO-like notation style of TTCN–3, the translated function bodies look very similar to the original. External functions can be easily implemented in C++ language.

Alternatives, Named alts. The C++ equivalents of TTCN–3 alt constructs are not obvious. Each branch of an alternative consists of three parts: a boolean guard expression, a guarding port or timer operation and the body. The translator routine of the compiler takes a sequence containing the translated form of the previous three parts and produces a single continuous C++ function block, which is equivalent with the original TTCN–3 alt construct.

Unlike the boolean approach of TTCN–3 operational semantics, the receiving and timeout operations can return one of three possible values. YES is returned if the operation was successful. MAYBE indicates that

the operation was not successful just now, but it can be succeeded in the future after some delay. NO is returned for an unsuccessful operation that can never be succeeded no matter how much time we wait. For example, a receive operation returns YES if the first incoming value matches the given template and NO if not. MAYBE is returned if the queue is empty.

The equivalent code block of alt constructs works according to the following algorithm. To avoid unnecessarily repeated guard evaluations, the C++ code defines a local flag variable for each branch to store the last result of guard evaluations. The algorithm iterates through the branches in an infinite loop and sets the flags according to Table 1.

Result of guard expression	Result of guard operation	New value of the flag
true	YES	YES
true	MAYBE	MAYBE
true	NO	NO
false	not evaluated	NO

Table 1. Rules for setting the alternative flags

If one flag becomes YES, the body of that branch will be executed and the alternative is finished. If all branches were visited and no successful one was selected, the algorithm checks the flags. If there will never be a suitable branch for selection, that is, all flags are set to NO, the test executor signals a dynamic testcase error. Otherwise, it blocks until something happens to ports or timers and it takes a new snapshot. Finally, the investigation of branches continues from the beginning, but only the guard operations are evaluated. Once a flag is set to NO, the guards of its branch will not be investigated again.

The TTCN–3 named alts are macros without own scope, thus they are translated to C preprocessor macros. These macros contain parts of the previous algorithm's body. The compiler translates expand statements to the expansion of these macros and the code of alt construct is glued together from its parts. Of course, this translation scheme can be applied on recursively nested alt constructs. The stand-alone receive or timeout statements are dealt with very similarly.

4. CASE STUDIES

This section shows some applications that demonstrate how our test executor can be used in real-life testing projects. Although our system has been used in more than ten – smaller or bigger – testing projects,

only the results of the most significant five conformance test suites are presented.

The first ATS covers the basic functionality of Internet Protocol version 6 (IPv6) [5] as well as its mobility extensions. The second test suite was written for testing the Management Information Base of IP routers supporting Simple Network Management Protocol (SNMP) [6]. The third example is an extensive conformance test suite for the IP based routing protocol Open Shortest Path First (OSPF) [7]. The fourth case study was developed for conformance testing the Robust Header Compression (ROHC) protocol [8], which can significantly reduce the overhead of IP headers for mobile terminals transmitting voice over IP. In our last example TTCN–3 was used both for conformance and performance testing of BRAIN Candidate Mobility Protocol (BCMP), which is a new IP micro mobility protocol.

Table 2 shows the sizes and characteristics of our example test suites. The translation and execution times were measured on a desktop PC with Intel Pentium III 450 MHz CPU and Linux operating system. We used GCC 3.0 as C++ compiler without optimization switches.

Projects	IPv6	SNMP	OSPF	ROHC	BCMP
Number of TTCN–3 modules	12	4	12	5	6
Total number of lines in ATS	12833	6395	17372	5777	3896
Number of data types	62	20	35	103	48
Number of templates	59	11	57	100	26
Number of test cases	205	10	181	44	41
Translation time (s)	587.9	68.4	164.0	139.1	50.3
Size of binary ETS (kbytes)	10867	804	2912	1807	710
Execution time (μs/operation)	162	217	515	1249	40

Table 2. Comparison of TTCN–3 test suites

The translation times include all steps required for getting the complete binary ETS from the TTCN–3 ATS. The most significant part of this time is spent on translating the generated C++ modules to binary code while the translation from TTCN–3 to C++ takes less than one second in all cases. The binary ETS includes the appropriate Test Ports and the Base Library as well. The relatively long compilation time and large ETS for IPv6 test suite is a result of the extensive use of named alts and expand statements.

The run-time performance of an ETS depends on the complexity of tested protocol. The most significant indicator is the time required for communication operations. The last row of Table 2 shows the average execution times of typical send operations, which include template build-

ing and evaluation, data encoding and sending on a physical interface. The main reason for the lower performance of the ROHC test suite is that it uses a relatively slow serial link as output device while the others send their data through a 10 Mbps Ethernet card. By publishing these results we want to set up some benchmarks for tool vendors and users to compare the forthcoming commercial TTCN-3 executors with.

5. CONCLUSION

This paper is the result of the development of a TTCN-3 test execution environment. It presents the system architecture of our test executor and outlines some implementation details. Our intention was also to share the experiences obtained within one year, during the usage of the tool and TTCN-3 language in different areas of testing. The run-time performance indicators of our executable test suites are also shown. We believe that our tool is fast enough for running performance tests on complex protocols as well.

In the future, we want to implement the missing or newly introduced TTCN-3 features in our tool, such as call based ports or regular expressions. On the other hand, we want to include semantic analysis and code optimization in our compiler. In addition, we do not agree completely with the frequent changes of TTCN-3 syntax and semantics that cause incompatibilities. This often requires the re-design of not only some parts of the test executor but the existing test suites as well.

References

[1] J. Grabowski, A. Wiles, C. Willcock, D. Hogrefe, On the design of the new testing language TTCN-3. In: Testing of Communicating Systems, volume 13, Kluwer Academic Publishers, 2000.

[2] Information technology – Open Systems Interconnection – Conformance testing methodology and framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++). ETSI TR 101 666 Version 1.0.0, May 1999.

[3] Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3. TTCN-3: Core Language. ETSI ES 201 837-1 Version 1.1.2, June 2001.

[4] T. Vassiliou-Gioles, S. Schulz, The TTCN-3 Runtime Interface (TRI); Concepts and Interface Definition of the TRI, ETSI Standard Draft, v4.2, September 2001.

[5] S. Deering, R. Hinden, Internet Protocol Version 6 (IPv6) Specification. RFC2460, IETF Network Working Group, December 1998.

[6] J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin, Simple Network Management Protocol (SNMP), RFC1157, IETF Network Working Group, May 1990.

[7] J. Moy, OSPF Version 2, RFC2328, IETF Network Working Group, April 1998.

[8] C. Bormann et al, RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC3095, IETF Network Working Group, July 2001.