



Georg-August-Universität
Göttingen
Institut für Informatik

ISSN 1611–1044
Nummer IFI–TB–2003–02

Technischer Bericht

The TTCN–3 module and template concepts revisited

Michael Schmitt, Michael Ebner

**Technische Berichte
des Instituts für Informatik
an der Georg-August-Universität Göttingen**

Mai 2003

Georg-August-Universität Göttingen
Institut für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14

Fax +49 (5 51) 39-1 44 15

Email office@informatik.uni-goettingen.de

WWW www.ifi.informatik.uni-goettingen.de

The TTCN-3 module and template concepts revisited*

Michael Schmitt¹

Michael Ebner²

May 2003

¹ Institute for Telematics e.V., Bahnhofstraße 30-32, DE-54292 Trier, Germany,
michael.schmitt@teststep.org

² Institute for Informatics, Georg-August-Universität Göttingen, Lotzestraße 16-18,
DE-37083 Göttingen, Germany, ebner@cs.uni-goettingen.de

The *Testing and Test Control Notation version 3* (TTCN-3) is a universal and standardized language for testing distributed systems. To open up new application areas for TTCN-3, proposals are made for controlling the execution of large sets of test cases, for the parallel execution of test cases, and for extended template matching mechanisms.

Keywords: Testing and Test Control Notation, TTCN-3, nested modules, parallel test execution, templates, matching mechanisms

1 Introduction

The *Testing and Test Control Notation version 3* (TTCN-3) [1] is a universal and standardized language for the specification and implementation of tests for distributed systems. TTCN-3 supports a broad spectrum of testing types, e.g., conformance and interoperability testing, and its communication mechanisms allow for testing various platforms such as the Common Object Request Broker Architecture (CORBA) or Internet-based protocols. Although TTCN-3 has reached a certain stage of maturity since its first release in 2001, we believe that some of its language elements could be further improved to make TTCN-3 applicable for even more application areas.

In this paper, three possible enhancements are proposed: In section 2, nested modules and remote module control part invocation are suggested. In section 3, a parallel operator

*This paper was accepted as a position statement for the 15th IFIP International Conference on Testing of Communicating Systems (TESTCOM 2003)

is described that allows to describe dependencies among test cases and execute them in parallel. These improvements are motivated by the way security testing is performed by scanners such as the open source tool `NESSUS` [2]. Security testing with `NESSUS` is characterized by three properties:

- A large number of test cases ($\gg 1000$).
- Strong dependencies among test cases (e.g., a mal-formed HTTP request is only sent if TCP port 80 has been identified as being open by a previous test case).
- Parallel test execution for single or multiple target systems.

Finally, the necessity for new matching mechanisms for records, arrays, and sets is motivated in section 4.

2 Nested Modules and Remote Control Part Invocation

A TTCN-3 document is composed of one or more *modules*. Each module represents either a complete executable test suite or a library. It consists of definitions and an optional *control part* that guides the execution of test cases. A module can import definitions from other modules but it cannot import their control parts.

Modules are the top-level structuring element in TTCN-3. Unfortunately, modules cannot be nested. Definitions can be combined into groups but a group does not define a new scope and has no semantic purpose except when definitions are imported by another module. Moreover, TTCN-3 assumes that the entire test execution is controlled exclusively by the control part of the current module and some auxiliary functions. However, in order to cope with large amounts of test cases, it is necessary to define and use several control parts.

Test case execution inside a module can only be structured by cascading function calls. For structuring purposes, one function could be defined for each module and group. It controls the test case execution of all test cases given inside this module or group. However, as mentioned above, groups do not define their own scope.

We suggest adding the group concept to the module concept to allow for nested modules.¹ Using nested modules also provides a more direct mapping between TTCN-3 and the Interface and Definition Language (IDL) which is used in CORBA [3]. Moreover, we suggest replacing the control part of modules by dedicated *control* functions to enable hierarchically structured control parts. By treating the control part as an ordinary function, it can also be called from detached modules.

The *control* function shall have no *runs on* clause. Thereby, configuration and communication statements are automatically forbidden. Control functions may be parameterized and return a value (e.g., a verdict) just like any regular function to provide more flexibility for test case execution.

The module name is used to access a control function from inside other control functions. To address control functions of inner modules, the dot notation of nested groups

¹We suggest keeping the group concept for structuring test suites on a loosely level.

Listing 1: Nested modules and control functions

```

1 module CGLabuses {
2   import from DenialOfService all;
3
4   function control() {
5     IIS (); PHP-Nuke();
6     execute(IIS.ExAir.ExAirSearch ()); // call a test case in a submodule
7   }
8
9   module IIS { // Internet Information Services
10    function control() {
11      var verdicttype v := pass;
12      DenialofService . IIS (); // call into a detached submodule
13      if (execute(DangerousSampleFiles) == pass) // samples installed?
14        v := ExAir ();
15    }
16
17    module ExAir { // Determines the presence of an ExAir ASP
18      function control() return verdicttype {
19        ...
20      }
21    }
22  }
23 }
24
25 module DenialofService {
26   module IIS { // Internet Information Services
27     function control() {
28       ...
29     }
30   }
31 }

```

is adopted. The example in Listing 1 shows the easy and seamless integration of nested modules and control functions into TTCN-3.

3 Parallel Execution of Test Cases

The selection of test cases and the order in which they are executed is specified in the module control part. Currently, the execution of a single test case blocks the execution of the control part, i.e., no further test cases can be executed in parallel. However, the sequential execution of test cases is too time-consuming for security testing. Hence, one should be able to specify which test cases can be run in parallel. For that purpose, some ideas from the UNIX tool MAKE [4] can be adopted.

3 Parallel Execution of Test Cases

An input file for MAKE consists of a number of dependency rules in the form

goal : [prerequisite] commands

A goal (typically a file) is achieved by running the corresponding commands. Before these commands can be executed, an optional prerequisite (dependencies) must be fulfilled which again may be one or more goals specified in another dependency rule. If the prerequisites of two or more goals are fulfilled, their commands can be executed in parallel.

For TTCN-3, we propose a parallel (**par**) statement which syntax resembles the one of TTCN-3's **alt** statement.

The general structure of the **par** statement is illustrated in Listing 2. In the given example, three goals – *info*, *mssql*, and *smurf* – are defined within the **par** statement. Each goal is followed by an optional number of repetitions (in square brackets; see below), its prerequisite (provided as a boolean expression in square brackets) and its commands (in terms of a statement block).

For convenience, the **par** statement defines implicitly an integer variable for each goal with the name as the goal and the scope of the **par** statement itself. Initially, all variables are set to 0. If the prerequisite of a goal is fulfilled, the corresponding commands are executed. After their termination, the goal variable is incremented implicitly.

In order to restrict the degree of concurrency, an optional parameter is suggested for the **par** operator that specifies the maximum number of parallel commands. It is specified in parentheses after the **par** keyword. In the given example, it is defined by variable

Listing 2: The **par** statement

```
1 module SecurityTests {
2   function control( integer maxNoOfTestCases, boolean denialOfService ) {
3     var verdicttype v := pass;
4
5     par ( maxNoOfTestCases ) {
6       info [ true ] {
7         v := execute( findServices ( ) );
8       }
9       mssql [ v == pass and info == 1 and port[1433] == open ] {
10        v := execute( MSSQLPasswords( ) );
11      }
12      smurf [10] [ v == pass and info == 1 and denialOfService == true ] {
13        v := execute( smurfAttack( ) );
14      }
15      else {
16        ...
17      }
18    }
19  }
20 }
```

4 Enhanced Matching Mechanisms

maxNoOfTestCases. By setting this parameter to 1, sequential execution is enforced while the test cases are still ordered automatically according to their prerequisites.

The execution of the `par` statement can be modeled by a scheduler that runs in parallel to all other processes. An important issue are the points in time at which the scheduler is active. There are two possible models, namely a nonblocked-scheduler and a blocked-scheduler model. The difference between both models is the frequency with which prerequisites are tested.

For maximum parallelism, the enabling conditions of the goals must be checked continually by a *nonblocked-scheduler*, since they may include variables that are modified by a *running* process. Checking prerequisites continuously may consume a lot of computation time. On the other hand, the boolean value of a prerequisite should only change at the end of another test case and thus, the *blocked-scheduler* process model can be adopted. If no enabling condition evaluates to true, the scheduler is blocked, even if the maximum degree of parallelism is not achieved. It wakes up as soon as the execution of the commands of some goal terminates.

Another design issue concerns the termination of the `par` as a whole, in particular in the case that the prerequisites of some goals cannot be fulfilled and there are no active processes any more. We suggest that an optional `else` goal is entered in the latter case. This allows to identify goals which have not been fulfilled by checking the goal variables.

In case that many identical tests are to be executed, the statement becomes unnecessary bulky if a separate goal has to be defined for each single test case instance. To resolve this problem, the `par` statement allows to specify how often a goal must be fulfilled. E.g., goal *smurf* in Listing 2 must be fulfilled 10 times. This mechanism also allows to execute n identical test cases in parallel if n is not known at compile time. If possible, the commands of an indexed goal are executed in parallel.

4 Enhanced Matching Mechanisms

TTCN-3 allows character patterns in templates to define the format of character strings. These character patterns have the same expressive power as regular expressions.

On the level of structured types, less powerful matching mechanisms are provided. In templates for arrays or sets/records of a single type, the matching operator “*” can be used as a wildcard for a sequence of zero or more elements. Example: { 1, *, 3 }. (Please note that TTCN-3 has two different interpretations for the “*” symbol: When used on the highest level inside an array, its meaning is *AnyElementsOrNone*, otherwise it means *AnyValueOrNone*.) However, there is no way to express, for instance, that a record with variable length shall consist of only one particular element. Listing all elements explicitly is not possible due to the unknown size of the record.

Hence, we propose to use the expressive power of regular expressions also for arrays as well as sets and records with elements of identical data type (*set of/record of*).

The following notation might be used for describing closures, groups, and alternatives: The term $x\#(min,max)$ inside a template matches with at least min and at most max occurrences of x where x is either a single element or a group of sequential elements.

5 Summary

Listing 3: Matching mechanism for structured types

```
1 var integer myInt := 8;
2
3 template record of integer RegExp :=
4   { <1, 2>#(2,5), 3#(,), 4, (5, 6, 7), myInt, ? };
```

Sequences of elements are grouped by `< ... >`. For the description of alternatives, the existing notation for value lists is used.

Listing 3 shows what regular expressions for arrays, records, and sets could look like in TTCN-3. Template `RegExp` would match a `record of integers` which consists of at least two and at most five 1-followed-by-2, zero or more occurrences of value 3, a single 4, either 5, 6 or 7, the value of variable `myInt`, and an arbitrary number.

5 Summary

In this paper, we have proposed several improvements to TTCN-3. The extensions of the module concept will require only moderate changes to the language and the parallel statement can even be integrated without invalidating existing TTCN-3 documents. The extensions for the template matching concept can be easily integrated, too. The support of testing *partially ordered* data values which is useful for, e.g., the Session Invocation Protocol (SIP), is left to future studies.

The authors are going to contribute their ideas to the standardization process at ETSI.

References

- [1] ETSI, European Telecommunications Standards Institute: ES 201 873-1 V2.2.1 — Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI, Sophia Antipolis, France (2002)
- [2] Deraison, R.: The Nessus Security Scanner. www.nessus.org (2003)
- [3] Ebner, M., Yin, A., Li, M.: Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In Schieferdecker, I., König, H., Wolisz, A., eds.: TESTING OF COMMUNICATING SYSTEMS XIV — Application to Internet Technologies and Services. Number 14, IFIP, Kluwer Academic Publishers (2002) 443–458
- [4] Free Software Foundation: GNU Make. <http://www.gnu.org/software/make/make.html> (2002)