Automated Maintainability of TTCN-3 Test Suites Based on Guideline Checking

George Din[1], Diana Vega[2], and Ina Schieferdecker[2]

[1] FOKUS Fraunhofer Institut, Kaiserin Augusta-Allee 31, Berlin, Germany
george.din@fokus.fraunhofer.de
[2] Technical University of Berlin, Franklinstr. 28-29
Berlin, Germany

Abstract. Similar to software development, the test development must be accompanied with a set of rules specifying how to write tests. They are grouped together into a document called guideline. Guidelines are especially necessary for large test specifications involving many developers and have the goal to reduce the effort of the overall development. So far, no universal guidelines for the TTCN-3 language [1] have been defined. Instead, each company or team defines and follows own development rules for test structuring and development. This paper deals with the problem of how to automate the validation whether a TTCN-3 test specification complies or not with an established guideline, i.e. guideline checking. The results of the validation process are a list of non-consistencies. A follow up step is the refactoring which automatically proposes and applies changes to improve the test suite compliance level, and thus its quality.

1 Introduction

In software engineering, guidelines may be defined for various aspects: models, programming, code documentation, users guides, developers guides, user interfaces, etc. They are useful for many reasons. First of all they help to establish a common understanding within the developing team. Next, they allow for easier development, changes or extensions. Any team member is able to understand the contributions of the rest of the team, and may even be able to extend parts contributed by other team members. Furthermore, new developers can integrate into the team by understanding much easier a complex system and being able to easily recognize its structure.

In this paper we consider the guidelines for test specifications written in the standardized Testing and Test Control Notation (TTCN-3) language [2]. We selected this language due to its popularity in the nowadays test developments. Its popularity grown over the last decade when many test suites have been specified in this language. Lots of resources have been invested by the industry and research groups in order to make out of TTCN-3 a general and standard testing framework. However, an important contribution to the spreading of TTCN-3 had the European Telecommunication Standardization Institute (ETSI)[3] which standardized various TTCN-3 test suites for telecommunication protocols.

Two obvious questions occur with respect to TTCN-3 based test specifications: on one hand, how well the tests are designed and, on the other hand, how to evaluate

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 417–430, 2008.

that they are well written in a consistent manner. Both questions can be answered by analyzing guidelines for TTCN-3 test development.

In testing area, the guidelines have the same importance as for software engineering. More specific we look into the problem of how to *automate* the guideline checking of test specifications and how to recognize potential non-consistencies with the specified guideline. To achieve that, we analyze several existent guidelines used for TTCN-3 test specifications. Then, we define a method to specify guidelines in such form that they can be used by an automated tool for guideline checking.

From a test quality perspective, the use of guidelines is an essential requirement. According to the quality model for test specifications proposed in [4], the guidelines compliance contributes the the overall quality of the test with respect to the selected quality criteria. In that model, the quality is seen as a set of characteristics; each characteristic being composed of further sub-characteristics. Several of these sub-characteristics may be evaluated in relation with guidelines:

- *understandability*: documentation and description of the overall purpose of the test specification are key factors in understanding a test suite.
- *learnability*: to be able to extend or modify a test suite, the test developer must understand how it is structured. Proper documentation or style guides have positive influence on learnability.
- *analyzability*: concerns the degree to which deficiencies in a test specification can be localized. For example, test specifications should be well structured to allow code reviews.
- *changeability*: describes the capability of the test specification to enable necessary modifications to be implemented. E.g. badly structured code or a test architecture that is not expandable may have negative impact on this.

One important question is how to check whether a guideline is fulfilled or not. As long as the nowadays software systems are very large and complex, the guideline checking should be automated as much as possible. Moreover, the guideline checking should not only determine whether an entity (e.g. documentation, program) is compliant with the guideline but also deliver a list of inconsistencies with precise localization information where the issues appear.

The information delivered by the guideline checker should then be used to fix the non-consistencies. The inconsistencies are of different types as for instance: a) *naming convention related*, e.g. a function does not start with `f_` , b) *logical*, e.g. a piece of functionality is placed in a wrong file, c) *structural*, e.g. a file is placed in a wrong package, etc. Also in this respect, we see the need for automation. This can be realized only on top of a taxonomy of types of inconsistencies which may appear. The automated approach should be such programmed that any type of inconsistency can be solved automatically or with very little human intervention.

The automation of guideline checking and inconsistencies solving should offer a tremendous help for rapid test specification improvement. An obvious result of automated approach is the better maintainability and reusability. This way the test specification can be specified in a consistent manner and changes can be easier propagated, etc. In addition, the same guideline can be used for different specifications belonging to the same application domain. Furthermore, the pieces of functionality (e.g. libraries)

specified according to a guideline are easier to be reused for another test specification that follows the same guideline.

This paper is structured as follows. The next section gives a short introduction of the TTCN-3 language. Section 3 looks in more detail into the structure of a guideline while Section 4 presents our method to define guideline checking rules and presents a classification of the refactoring possibilities. The guidelines of the IPv6 testsuite[5], written in TTCN-3, are provided as example and discussed in Section 5. The paper finishes with the overview on related work and the conclusion sections.

2 A Short TTCN-3 Overview

The TTCN-3 language is a text-based language and has the form of a modern programming language, which is obviously ease to learn and to use. Specially designed for testing, it inherits the most important typical programming language artifacts, but additionally it includes important features required for test specification.

A TTCN-3 based test specification is called Abstract Test Specification (ATS) and it usually consists of many files grouped into folders and subfolders. Each file contains one or more **modules**. The **module** is the top level element of the TTCN-3 language which is used to structure the *test definitions* of:

- test data: **types** of messages, instances of types called **templates**,
- test configurations: **ports** and **components** to define the active entities of a test,
- test behaviours: **functions**, **altsteps**, **testcases** which implement the interactions between the **components** and the SUT and which make use of the test data,
- control: a global behavior to control the flow of **testcases** execution

Each afore mentioned definition type (except control which does not need an identifier)) has an identifier and can be placed in any module. TTCN-3 also offers the possibility of grouping elements into **groups**. The test developer is free to choose how to name the identifiers, how to group the definitions and in which modules to place them. However, the **group** element does not impose a new scope for the grouped elements, but only at the logical and visual level.

As long as the test specifications contain thousands of definitions, it is extremely important to be consistent in writing TTCN-3 test definitions and in maintaining a clear test suite structure and file structure. The language is similar to a programming language such as Java or C++, therefore lots of structuring possibilities, naming conventions etc. are allowed.

Currently, all ETSI test specifications are written in the TTCN-3 language [6]. Along the last decade, the ETSI test specifications adopted different guidelines for structuring, naming conventions etc. We are interested in this evolution in the testsuite design and try to derive a general view on guideline rules design.

We analyzed a number of test specifications (SIP[7], IPv6[5], M3UA[8]) standardized by ETSI in order to learn which guidelines have been used and check how consistent are they along the whole test specification. A classification of these rules is realized in Section 3 while in Section 4 we present a method of how to describe guideline rules such that an automated guideline checker is capable of checking the test specification consistency with respect to those rules.

3 Guideline Rules Classification

A comprehensive guideline should take into account various aspects. We propose a method to classify these aspects for TTCN-3 into three levels: *physical level*, *language level* and *architectural level*. Guideline rules are defined at each level and, consequently, they contribute with requirements to the global guideline. Fig. 1 illustrates these levels.

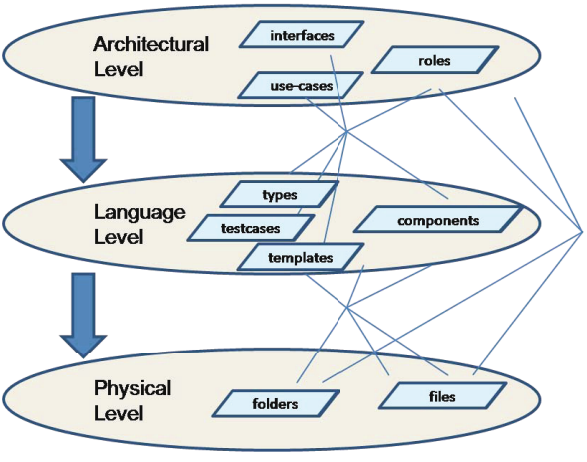


Fig. 1. Guideline Levels

The architectural level refers to information related to System Under Test (SUT) (interfaces, use-cases, roles etc.), the language level refers to the definition of test constructs in the TTCN-3 language (types, components, testcases, etc.), while the physical level deals with file system information such as files and folders. In this classification, the information from one level may propagate only to the levels below (top-done) and never to the above ones. We analyze these levels in greater detail.

3.1 Architectural Level

The architectural level includes guideline rules derived directly from the SUT architecture.

- *SUT interfaces*: the interaction between the Test System (TS) and SUT is realized over at least one interface. To increase the readability, a common guideline rule is to group together the definitions related to one interface.
- *roles*: the test behavior can be designed for different roles, e.g. client, server, proxy. The test definitions defined for one role should be grouped together.
- *use cases*: a test behavior corresponds to a type of interaction, i.e. use case, with the SUT. Multiple use cases can be treated within the same test specification. To avoid mixing the test behaviours from different use cases, a common practice is to group together the definitions related to a use case.

– *version*: a test specification may refer to multiple versions of the tested SUT' specification. A common practice is to avoid that test definitions for different SUT versions are mixed.

The information from the architectural level is used to structure the test specification and, consequently, imposes guideline rules to the two levels below. At the language level, the architectural information is used to group related test definitions into TTCN-3 modules and groups. Additionally, naming conventions can also be used in order to embed architectural information into the TTCN-3 identifiers.

We give as example an SUT which has two interfaces: Interface1 and Interface2. An architecture level guideline rule should say that the test definitions related to each interface should be placed in the same group. At the logical level we have several options to propagate the architectural rule. We may either define two TTCN-3 modules or two groups. In either case the definitions will be grouped together: `Interface1_Definitions_Module` (or Group) and `Interface2_Definitions_Module` (or Group). A further option is to use also naming conventions for the involved TTCN-3 types, templates or testcases such as `tc_interface1_Test1`, `tc_interface1_Test2`, where the prefix `tc_` stands for testcase abbreviation. However, the three posibilities can be combined. For instance, the `tc_interface1_Test1` can be added to a group of testcases for Interface1 `Interface1_Testcases_Group` which is defined in a module named `Interface1_Definitions_Module` as illustrated in Listing 1.1.

Listing 1.1. Test Structuring Example

```
1  module Interface1_Definitions_Module {
2     group Interface1_Testcases_Group {
3        testcase tc_interface1_Test1
4              runs on C system S {
5           . . .
6        }
7        testcase tc_interface1_Test2
8              runs on C system S {
9           . . .
10       }
11    }
12 }
```

At the physical level, the architectural information is used to store the test definitions into files and folders. Also in this case, naming convention rules can be used to name the files and folders. Following the example provided above, we can store all definitions related to each interface into separate folders such as: types/interface1/File1.ttcn3, components/interface1/File2.ttcn3, etc. When more than one architectural guideline rules apply, they can be combined in an arbitrary order.

3.2 Language Level

The language level contains guideline rules for writing the TTCN-3 code. They can be classified into:

- *formatting rules* related to indentation, braces, white spaces, blank lines, new lines, control statements, line wrapping and comments
- *naming conventions* related to the names of the identifiers of the TTCN-3 constructs (types, templates, testcase, components, etc.)
- *structural rules* related to grouping the test definitions into groups and modules.

The naming conventions concern prefixing (and sometimes postfixing) rules and apply to all TTCN-3 elements which require an identifier: types, templates, functions, altsteps, testcases, groups, modules, variables, etc. For easier localization, the TTCN-3 identifiers can be prefixed with a string indicating a group of definitions of the same category. For example, the message types can be prefixed by strings such as `Type`, `type`, `type_`, `T_` etc. Multiple prefixes can occur. For example, type definitions can be grouped into types of messages to be sent to SUT, e.g. `Send_Msg`, and types of messages to be received, e.g. `Received_Msg`. If multiple prefixes are used, they can simply be concatenated or separated by the "`_`" character.

The structural rules concern the grouping of the definitions into groups and modules. This can be realized in many ways:

- *grouping by categories*: the definitions of the same category can be grouped together (e.g., types in a group of types, templates in a group of templates).
- *grouping by libraries*: the reusable definitions which are at the same time also general enough to apply to different test suites should be grouped into libraries.

3.3 Physical Level

The physical level offers further structuring possibilities of TTCN-3 definitions:

- files: store particular groups of definitions in separate files
- folders: files can be further grouped into folders and subfolders.

Also at the physical level the naming conventions should appear. They are usually propagated from the upper levels and impose prefixes for the names (or even impose the name itself) of files or folders. For example, a file located as `/types/interface1/usecase1/sending.ttcn3` combines information from the architectural level i.e. `interface1` and `usecase1` with information from the language level i.e. `types` and `sending`. This file name means that it contains all types of messages to be sent to SUT defined for `usecase1` and for `interface1`.

4 Test Analyzability and Refactoring

To ensure that a guideline is followed consistently along the whole test specification, a *guideline checker* is needed. *Test analyzability* is the characteristic of a test to be validated against a guideline and it includes the mechanisms to define and check guideline rules. *Refactoring* is the mechanism which enables to fix inconsistencies detected during the analyzability phase.

4.1 Guideline Checker Types

Guideline checking implies that all guideline rules are verified on top of a test specification. Our realization approach is illustrated in Fig. 2. The guideline rules are all managed by a common repository and are loaded by the guideline checker. Another input of the checker is the test specification itself. The guideline checker consists of rule checkers which are of different types. Moreover, each checker type can be instantiated for an arbitrary number of times (one instance per guideline rule). The checker reports for each rule how many identifiers matched that rule and how many of them did comply with it.

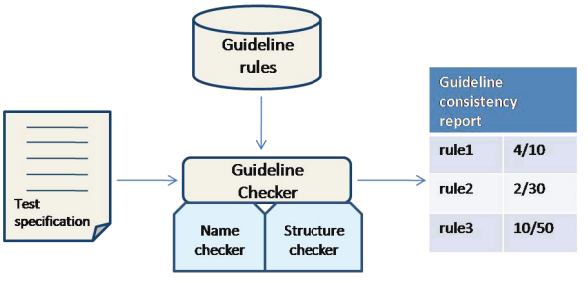


Fig. 2. Guideline Rules Checkers

We identify two types of checkers: *naming conventions checkers* and *structural checkers*. A guideline rule is instantiated in one of these checker types and it is applied to all identifiers of a test specification. The naming convention checkers evaluate the name of the identifier and determines if it is composed correctly. The structural checkers verify whether the test definition whose identifier is evaluated is placed in the correct structure (group, subgroup, module, file and folders).

4.2 Checking Rules Specification

A guideline rule consists of three parts: a *filtering criterion* which indicates which identifiers should follow the rule, a *relation* and an *entity*. The relation and the entity define *what* the test definition selected by the filtering criterion *should comply with*.

Table 1. Rules examples

Rule1	*(testcase)(naming:prefix)(tc_)*
Rule2	*(testcase)(inclusion:module)("Testcases")*
Rule3	*(testcase)(naming:prefix)(arch_info:interface)*
Rule4	*(testcase)(inclusion:group)(arch_info:use-case)*

Fig. 3 depicts the structure of a rule. There are two types of relations: *naming relations*, which define how the identifiers should be created, and *inclusion relations*, which describe where to place a test definition into a structural element (group, module or file).

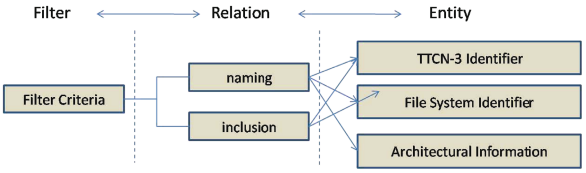


Fig. 3. Rule Specification

The *entity* can be a TTCN-3 identifier of a structural construct (group or module), a non-structural TTCN-3 construct (component, testcase, etc.), a file system identifier (of a file or folder) or an architectural information (role, interface, version or use case).

As shown in the figure, an *inclusion* relation is not possible between an identifier and an architectural information. The *naming* relation is possible with an architectural information since an identifier can be prefixed with such information.

To illustrate how these rules are created we provide a few examples in Table 1. An identifier should match all rules which apply to it in a top-down order. All four rules defined in the table have as filtering criterion the TTCN-3 construct "testcase" and means that all testcases in the test specification should comply with these rules. The Rule1 and Rule3 concern prefixing information which means that a valid testcase identifier should be prefixed with the information provided in the rule's *entity*. A valid testcase identifier is `tc_Interface1_test1` since it is prefixed first with `tc_` according to Rule1 and with `interface1` according with Rule3. The `tc_test2` is not correct since it does not comply with the Rule3. The second rule says that all testcases should be defined within the module with the name "Testcases" given as a string. The forth rule requires that the testcases are grouped into groups which have names derived from use-cases names.

4.3 Refactoring

Refactoring has been discussed in detail in [9]. For testing, refactoring is defined similar to software engineering refactoring, as the manual or automated improvement of the structure of a test specification. There are many types of refactorings we can encounter in a test specification. We highlight here the most used ones:

- *formatting*: implies indentation and changes of the locations of test definitions in a file in terms of lines and columns.
- *renaming of identifiers*:gives the possibility to rename an identifier (TTCN-3 language element, file name, etc). Some parts of the identifier (e.g. if the identifier should be prefixed by the module name but it is not) can be changed in an automated way. The refactoring task should also change all references to that identifier in the associated visibility scope. This type of refactoring is used for situations when an identifier does not follow a naming convention rule, as for instance: a component type should be prefixed by `CT_` or should start with capital letter but it does not.
- *moving a definition into another group*: we distinguish between moving an identifier into a group in the same module or into a different module or file. The latest two cases fit into the next refactoring schemas since they affect the module

importing and file inclusion settings. If an identifier is moved to a group within the same module, the refactoring mechanism has to take care whether the identifier name should be prefixed by the group name. This type of refactoring is needed to handle inconsistencies such as, for instance, a component type definition is not placed in the group which should contain component definitions.
- *moving a definition into another module*: in this case, the moved test definition has to be imported in the modules where it is referred by using the **import** construct. Also in this case, the refactoring has to be consistent with the naming conventions regarding identifier prefixing.
- *moving a definition into another file*: has the same constraints as the case of moving an identifier to another module (moving a test definition to another file implies moving to another module as well) but also impacts the file inclusion settings for the whole project with respect to compilation.

Sometimes, for a given non-consistency, more than one refactoring possibilities may apply. In these situations the manual intervention is required.

Many refactoring rules can be derived from software engineering [9] and applied to TTCN-3 as presented in [10]. However, our aim was not to identify all of them but rather to develop a method to classify the guideline rules on various levels (architectural, language and physical) and understand how they propagate from one level to another. The refactoring schemes are only example of how non-consistencies can be handled in an automated manner.

5 An Example - The IPv6 Test Suite

We selected for our analysis the standardized TTCN-3 test suite IPv6 [5] published and free to download from ETSI web site [3]. Test specifications for IPv6 protocols are foreseen to cover both conformance testing and interoperability testing for IPv6 core protocols (such as IPv6 specification, neighbor discovery and stateless address auto-configuration) and extended protocols (such as security, mobility, and transition).

5.1 Test Specification Analysis

Architectural and Physical Level Guidelines Analysis. Fig. 4 shows a view of how the ATS has been structured at the physical level. Three important guideline rules have been applied at this level:

- *folder structuring guideline*: First, the TTCN-3 files which belong to a common logical functionality are grouped together. This structure combines an architectural level rule with the physical level and it is reflected in the existence of two types of folders: a) with common functionality, i.e. library folders such as `libCommon`, `Libcore`, etc. and b) with specialized functionality, e.g. `AtsCommon`, `AtsCore`.
- *folder/file/module naming convention guideline*: Two guideline rules have been applied in top-down order. The first rule regards the association between a file and a folder. It is reflected by the naming convention which requires that the file name has to start with the name of the folder that contains that module,

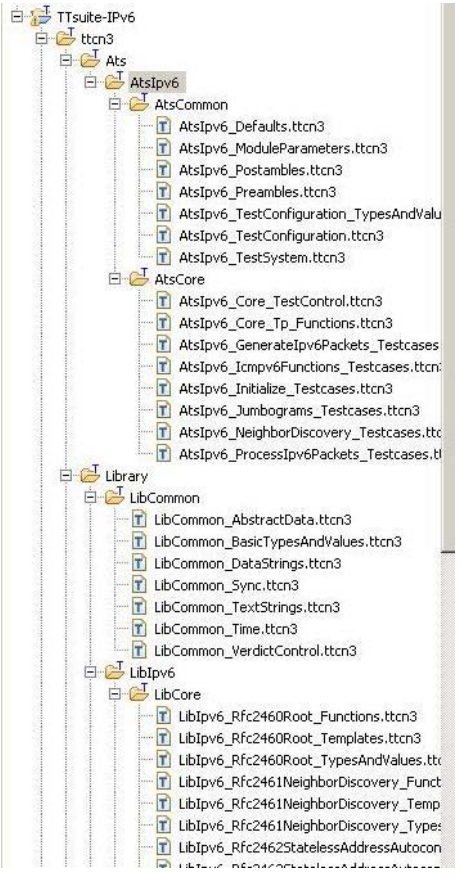


Fig. 4. IPv6 Physical Level Guidelines

e.g. `LibCommon_AbstractData.ttcn3` is placed in the folder `LibCommon`. The second rule specifies that the file name has to encapsulate the description of the predominant type of TTCN-3 elements enclosed in the analyzed module.

– *structuring based on architecture information*: The testcases have been grouped in files caring the names of the use cases such as initialization, neighbor discovery: `AtsIpv6_Initialize_Testcases`, `AtsIpv6_NeighborDiscovery_Testcases`, etc.

With respect to guideline checking, the validation of the first guideline is difficult to automate since it is not possible to decide which functionality should belong to a library. However, the second and the third guidelines can be checked in an automated manner since they only verify the established naming or inclusion convention.

Language Level Guideline Compliance. There are many naming conventions used in this specification. We provide, as example, the naming convention for the behavioral names. These are based on the rule:

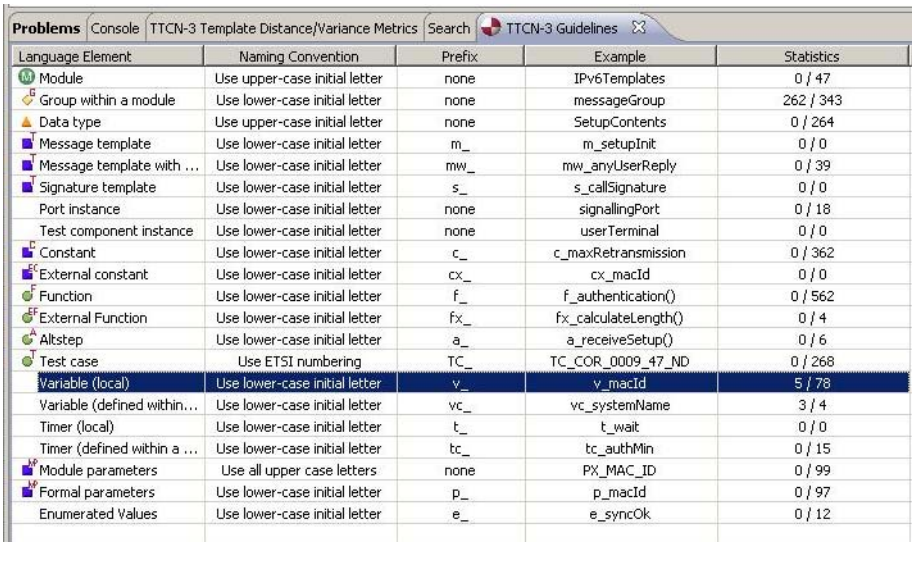
```
  <protocol>_<main functionality>_<role>
_<functionality>_<type>_<nnn>
```

The <protocol> is the IPv6 specification (IP6). <main functionality> separates definitions by protocol header type into Header (HDR) and Extension header (EHR). The roles are also used for classification. <role> is one of following: Host (HST), Router (RTR), Node (NOD), Source Host (SOH) and Destination Host (DEH). The <functionality> is used to classify test definitions by use case into General (GEN), hop-by-hop options header (HBH), destination options header (DSH), routing header (ROH), fragmentation header (FRH) and IPsec headers (SEC). <type> further classifies test definitions into Valid Behavior, Invalid Behavior, Inopportune Behavior and Timers (TI). The <nnn> is a simple sequential number between (001 999) to distinguish different tests of the same category.

Many naming conventions rules regarding other TTCN-3 constructs are presented in Fig. 5 together with their statistics.

5.2 Implementation

For the automation process of guideline checking we provide an implementation based on the TTworkbench [11] tool, an Eclipse-based IDE that offers an environment for specifying and executing TTCN-3 tests. The main reason for selecting this tool, is that it provides a metamodel for the TTCN-3 language which is technically realized on top of Eclipse EMF [12]. EMF is a Java framework and code generation facility which helps turning models rapidly into efficient, correct, and easily customizable Java code.



Language Element	Naming Convention	Prefix	Example	Statistics
Module	Use upper-case initial letter	none	IPv6Templates	0 / 47
Group within a module	Use lower-case initial letter	none	messageGroup	262 / 343
Data type	Use upper-case initial letter	none	SetupContents	0 / 264
Message template	Use lower-case initial letter	m_	m_setupInit	0 / 0
Message template with ...	Use lower-case initial letter	mw_	mw_anyUserReply	0 / 39
Signature template	Use lower-case initial letter	s_	s_callSignature	0 / 0
Port instance	Use lower-case initial letter	none	signallingPort	0 / 18
Test component instance	Use lower-case initial letter	none	userTerminal	0 / 0
Constant	Use lower-case initial letter	c_	c_maxRetransmission	0 / 362
External constant	Use lower-case initial letter	cx_	cx_macId	0 / 0
Function	Use lower-case initial letter	f_	f_authentication()	0 / 562
External Function	Use lower-case initial letter	fx_	fx_calculateLength()	0 / 4
Altstep	Use lower-case initial letter	a_	a_receiveSetup()	0 / 6
Test case	Use ETSI numbering	TC_	TC_COR_0009_47_ND	0 / 268
Variable (local)	Use lower-case initial letter	v_	v_macId	5 / 78
Variable (defined within...	Use lower-case initial letter	vc_	vc_systemName	3 / 4
Timer (local)	Use lower-case initial letter	t_	t_wait	0 / 0
Timer (defined within a ...	Use lower-case initial letter	tc_	tc_authMin	0 / 15
Module parameters	Use all upper case letters	none	PX_MAC_ID	0 / 99
Formal parameters	Use lower-case initial letter	p_	p_macId	0 / 97
Enumerated Values	Use lower-case initial letter	e_	e_syncOk	0 / 12

Fig. 5. Guidelines Rules Compliance View

Our work on the automated guideline checker follows up an earlier work [13] where TTCN-3 test quality indicators are derived from a static analysis of a TTCN-3 test suite, i.e. only the test sources are need. This is different from a dynamic analysis where the investigations regard the test execution as well. The implementation is designed as a plug-in whose invocation triggers the following actions:

- access the EMF metamodel instance of the TTCN-3 test specification
- traverse and correlate the elements of interest
- validate the guidelines and store the results
- refactor the whole test specification according to guideline rules.

We implemented and applied the set of ETSI TTCN-3 naming conventions [14] on the Ipv6 test specification. An intuitive guideline compliance statistic is always welcome by test developers and has the advantage of a rapid identification of the issues in the test specification. Therefore, we choose the tabular presentation format encapsulated in a new Eclipse View. Fig. 5 presents the applied naming convention and what level of compliance has been achieved, i.e. in a statistical manner:

$$ComplianceLevel = \frac{\textit{No of non respectig elements}}{\textit{No of elements}}$$

Each line in the table corresponds to a rule and consists of a) the expression-pattern that the name has to follow, b) an example, and c) the obtained statistic. The list of non-consistencies can be visualized in a separate window presenting the identifiers of non-conforming entities. Each identifier can be replaced with a new one; the refactoring process behind will refactor the new name along the whole test suite.

Looking into the results, we notice that the ETSI IPv6 test suite respects integrally the naming conventions except the ones related to the *group* and variable names. As the Fig. 5 indicates, 262 out of 343 groups do not respect the convention of lower-case initial letter.

With respect to refactoring, the user has then the possibility to select one of the rules which are not entirely fulfilled, e.g., the rule selected in Fig.5 is not satisfied by 5 identifiers out of 78. Next, the GUI provides the list of inconsistencies for that rule. For each inconsistency the user is asked to introduce a new identifier. By applying the new modifications, the old identifier is replaced with the new introduced one within the whole test suite.

6 Related Work

The guidelines are designed to help the developer in writing better code. They are available for almost any programming language and have impact on different levels such as: coding level, for instance for C++ in [15], design level, for instance for Java in [16], formatting level, comments level, etc.

On the testing side, the existing work focuses more on the guidelines regarding the effectivity of various types of tests: unit tests, integration tests, system tests etc. The work in [17] highlights a set of 27 guideline rules for writing jUnit tests.

With respect to TTCN-3, reusability has been explored in [18]. This work concentrate in great detail on guidelines for writing reusable TTCN-3 code. Maintainability

aspects, and in particular refactoring, have been concerned in [19] where catalog of 20 refactoring rules derived from Java [9] have been proposed and implemented. Refactoring is seen as a technique to systematically restructure code to improve its quality and maintainability while preserving the semantics.

7 Conclusion

In this paper we introduced, analyzed and classified TTCN-3 test specification guidelines. In order to investigate and identify the compliance to guidelines, a reverse engineering mechanism is needed. The automation of this process is essential as long as the nowadays test specifications consists of thousands of test definitions.

The introduced concepts ensure a structured and rule oriented thinking of guidelines. The novelty of this approach relies on identifying the levels of guideline rules for TTCN-3 test specifications. Additionally, we take into account the rule propagation from one level to another.

We foresee several possible extensions. The guideline rules can be extended to further rules such as: ontology based naming conventions, code documentation, etc. Another idea to be explored in future work is the combination of guideline rules obtained from the architectural information with test definition generation. This will make possible the systematic generation of test specification skeletons from a minimal information about the SUT (interfaces, use cases, roles, versions).

References

1. ETSI: Etsi standard es 201 873-1 v3.1.1 (2005-06): The testing and test control notation version 3; part 1: Ttcn-3 core language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2005)
2. Willcock, C., Dei, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: An Introduction to TTCN-3. John Wiley & Sons, Ltd, Nokia Research Center, Nokia, Germany, Nokia, Finland (April 2005)
3. ETSI: European Telecommunication Standards Institute - ETSI
4. Zeiß, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 Quality Model to Test Specifications Exemplified for TTCN-3 Test Specifications. In: Software Engineering 2007 (SE 2007), March 2007. Lecture Notes in Informatics (LNI), Copyright Gesellschaft für Informatik, Köllen Verlag, Bonn (2007)
5. European Telecommunication Institute - ETSI: Internet Protocol version 6 (IPv6) Conformance Test Specification (2006)
6. Wiles, A.: ETSI testing activities and the use of TTCN-3 (2001)
7. European Telecommunication Institute - ETSI: Session Initiation Protocol (SIP) Conformance Test Specification (2006)
8. European Telecommunication Institute - ETSI: MTP Level 3 User Adaptation Layer (2002)
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
10. Zeiß, B.: A Refactoring Tool for TTCN-3. Master's thesis, Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-BM-2006-05, ISSN 1612-6793 (Tippfehlerbereinigte Version), Zentrum für Informatik, Georg-August-Universität Göttingen (March 2006)

11. TestingTechnologies: TTworkbench: an Eclipse based TTCN-3 IDE, http://www.testingtech.de/products/ttwb_intro.php
12. Eclipse: Eclipse Modeling Framework (EMF) (2008)
13. Vega, D.E., Schieferdecker, I.: Towards quality of TTCN-3 tests. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320. Springer, Heidelberg (2006)
14. ETSI: ETSI Naming Conventions (2007)
15. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, Reading (1986)
16. Sun Microsystems, I., Javasoft: Java Look & Feel Design Guidelines. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
17. Services, G.S.: Unit testing guidelines (2007)
18. Mäki-Asiala, P.: Reuse of ttcn-3 code. Master's thesis, VTT Electronics Helsinki (2005)
19. Zeiß, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P.: Refactoring and Metrics for TTCN-3 Test Suites. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 148–165. Springer, Heidelberg (2006)