RESEARCH PAPERS
FACULTY OF MATERIALS SCIENCE AND TECHNOLOGY IN TRNAVA
SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

2014 Volume 22, Special Number

SPECIFICATION-BASED TESTING VIA DOMAIN SPECIFIC LANGUAGE

Michal SROKA[1,2], Roman NAGY[2], Dominik FISCH[2]

ABSTRACT

The article presents tCF (testCaseFramework) - a domain specific language with corresponding toolchain for specification-based software testing of embedded software. tCF is designed for efficient preparation of maintainable and intelligible test cases and for testing process automation, as it allows to generate platform specific test cases for various testing levels. The article describes the essential parts of the tCF meta-model and the applied concept of platform specific test cases generators.

KEY WORDS

Specification-based testing, Domain specific language, Abstract test case, Test design efficiency

INTRODUCTION

Software testing is a key part of the software development process and essential part of quality assurance, also in software development of car basic functions. Specification-based testing is a necessary part of our verification process, as it is required to guarantee[2] the correct behaviour of the developed software.

As we deal with considerable amount of test cases, their maintainability, intelligibility and preparation efficiency is important. Therefore we developed tCF (testCaseFramework), what is a domain specific language (DSL) for writing test cases providing the properties mentioned above. In addition to the DSL, tCF also includes Xtext-based (Eclipse Foundation, 2013) Eclipse editor with advanced usability features, for example code completion, content assist, code validation, outline, syntax highlighting, file independence and built-in platform specific test code generator.

[1] Fixing costs of any discovered error after the software product deployment are usually tens to thousands times higher than fixing costs of any error discovered during development (Baziuk, 1995). Moreover, it can negatively influence the image of the car brand.

Michal SROKA[1,2], Roman NAGY[2], Dominik FISCH[2]
[1] Institute of Applied Informatics, Automation and Mathematics, Faculty of Materials Science and Technology, Slovak University of Technology in Bratislava, Paulínska 16, 917 24 Trnava, Slovakia
[2] Car Basic Functions Software Development, Research and Development Centre, BMW AG, Knorrstraße 147, 80807 Munich, Germany
michal.sroka@stuba.sk, roman.nagy@bmw.de, dominik.fisch@bmw.de

The following sections will describe the designed DSL and also the abilities of the corresponding editor-based tool.

USE-CASE BASED ABSTRACTION

In specification-based testing, the test cases are designed in pursuance of software requirements. Since software requirements are written in form of use-cases, the same level of abstraction should be applied also in test cases. Every test case providing use-case level of abstraction can be accomplished in a car (Nagy, 2013).

To reach this level of abstraction, actors and their values (states) need to be defined in tCF:

- **Actor** is a single settable and/or checkable element in the car, for example light switch, headlight, remote control;
- **Actor's value** is a state, to which the actor is being set or which is to be checked on the actor, for example actor light switch can be in positions off, on, automatic or parking; headlight can be in state on, off or dimmed.

Actors and their values, as they were specified above, are basic elements of use-case based abstraction. In a test case, actors can be set to one of the available states or the actor can be checked if it is in one of the available states.

ABSTRACT TEST CASE IN tCF

The basic abstraction of the tCF DSL was described in the previous section. This section specifies a **test case** and its elements. A tCF test case consists of sets, checks, time steps and calls to steps. Those elements are described in the following paragraphs.

Sets and **checks** are tCF elements, which connect an actor with one of the available values in order to provide an input to the tested software or to check an output of the tested software. The relation of actors, their values, sets and checks is clarified in Fig. 1.

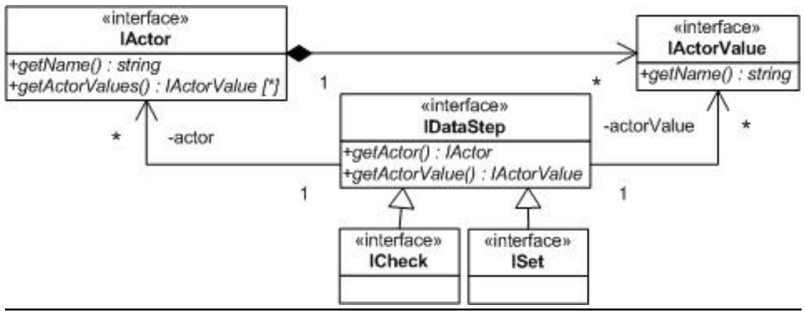


Fig. 1 *Actors, ActorValues, Set and Check relation*

As the developed automotive software runs on electronic control units (ECU), it is also necessary to specify **time steps** between sets and checks.

There is also the possibility to specify named sequences of elementary features (set, check, time step), which is called in tCF **step**. A step corresponds to a procedure in terms of programming languages. An example of an often applied step is the usage of the remote control in order to unlock a car: the user normally presses the button, keeps it pressed for a few milliseconds and then releases the unlock button (that are three elementary actions taken by the user to unlock the car).

The overall test case interface description is shown in *Fig. 2*.

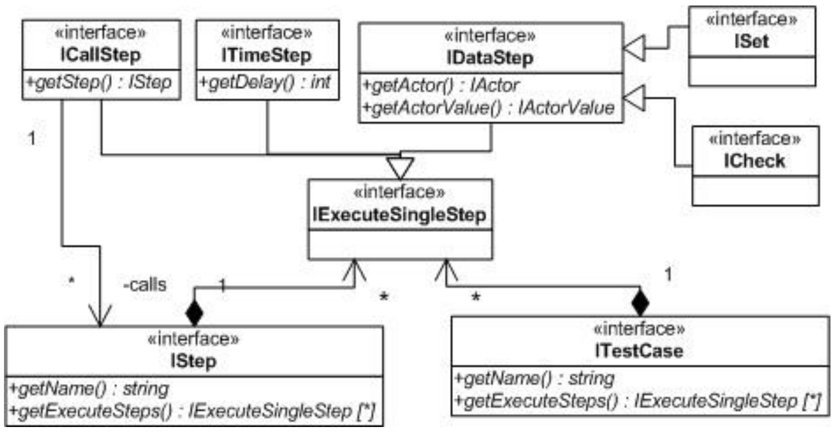
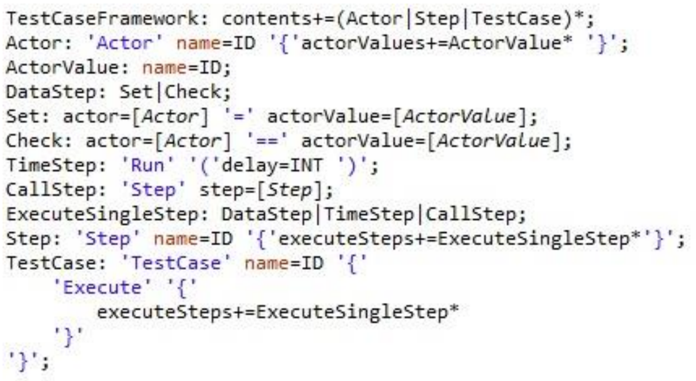


Fig. ***2*** *tCF test case structure*

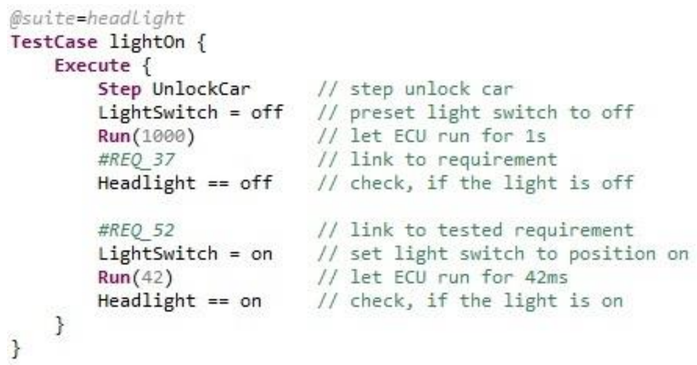
As we have specified the necessary entities in tCF, we can provide also the grammar of DSL (Code **1**). The grammar describes the syntax of entities present in the designed language.



```
TestCaseFramework: contents+=(Actor|Step|TestCase)*;
Actor: 'Actor' name=ID '{'actorValues+=ActorValue* '}';
ActorValue: name=ID;
DataStep: Set|Check;
Set: actor=[Actor] '=' actorValue=[ActorValue];
Check: actor=[Actor] '==' actorValue=[ActorValue];
TimeStep: 'Run' '('delay=INT ')';
CallStep: 'Step' step=[Step];
ExecuteSingleStep: DataStep|TimeStep|CallStep;
Step: 'Step' name=ID '{'executeSteps+=ExecuteSingleStep*'}';
TestCase: 'TestCase' name=ID '{'
    'Execute' '{'
        executeSteps+=ExecuteSingleStep*
    '}'
'}';
```

Code 1 tCF grammar in Xtext

Now, when the structure of the tCF test case is described and specified, an example of a test case can be provided. Assume having the following (artificial and simplified) requirement with ID REQ_52: „*When the car is unlocked and the light switch is turned to position on, the headlight is switched on.* “ The corresponding test case is in Code **2**.



```
@suite=headLight
TestCase lightOn {
    Execute {
        Step UnlockCar       // step unlock car
        LightSwitch = off    // preset light switch to off
        Run(1000)            // let ECU run for 1s
        #REQ_37              // link to requirement
        Headlight == off     // check, if the light is off

        #REQ_52              // link to tested requirement
        LightSwitch = on     // set light switch to position on
        Run(42)              // let ECU run for 42ms
        Headlight == on      // check, if the light is on
    }
}
```

Code 2 Example of simple test case

This kind of test case description is well maintainable, intelligible and easy to write, what is one of the important advantages of DSL usage. The abstraction of tCF test cases brings also another advantage: it is implementation independent; therefore multiple implementations can be generated out of this single test case specification. This will be further described in the following sections.

There are two features in the example test case (Code **2**), which were not explained yet:

- Links to requirements. The element starting with a sharp (#) specifies a link to a tested requirement. In specification-based testing, it is often important to watch requirements coverage and tCF can do it automatically.
- Test annotation. It is possible to set additional features of the test case, for example a test suite. This allows grouping and filtering test cases to be executed.

MODEL-BASED APPROACH VIA ALTERNATIVES

Yearlong experiences showed that a significant amount of our requirements could be tested more efficiently on the specified abstraction level. For example, consider the following requirement: „*When the car is unlocked and the light switch is turned to position on or to position automatic, the headlight is switched on.* “

With the above described structure, it would be necessary to test this requirement in two test cases, as there are two options of light activation.

Therefore we designed elements named alternatives in tCF. The class diagram is shown in *Fig. 3*.

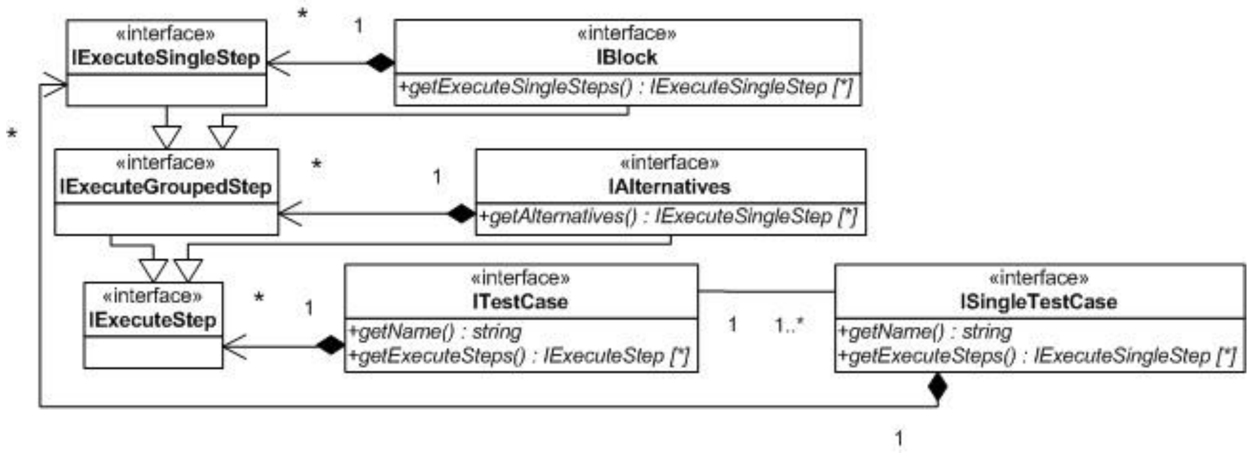
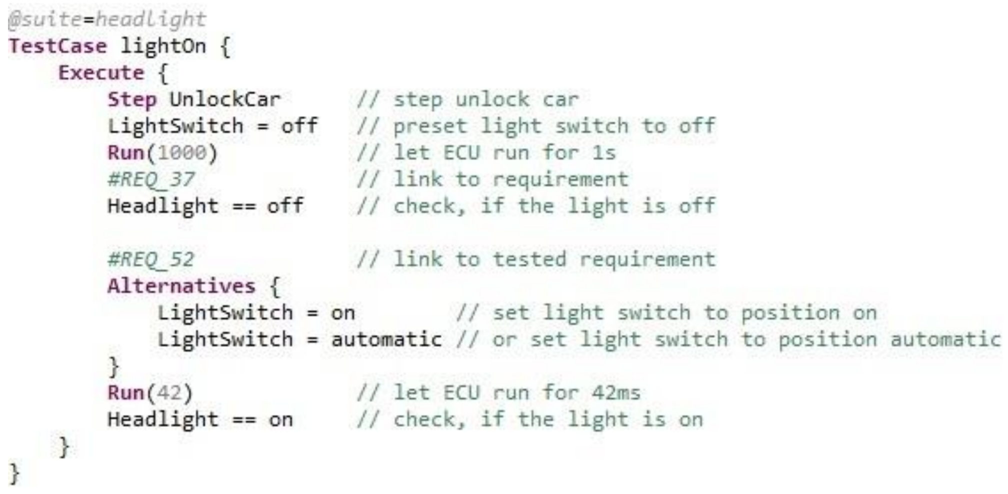


Fig. 3 *Alternatives software design*

The test case from Code **2** can then be changed such that it represents both tested cases. The adjusted test case is shown in Code **3**. From this single abstract tCF test case will be two non-alternative test cases generated (the first will put light switch to position on and the second to position automatic in order to activate headlight).



```
@suite=headlight
TestCase lightOn {
    Execute {
        Step UnlockCar      // step unlock car
        LightSwitch = off   // preset light switch to off
        Run(1000)           // let ECU run for 1s
        #REQ_37             // link to requirement
        Headlight == off    // check, if the light is off

        #REQ_52             // link to tested requirement
        Alternatives {
            LightSwitch = on        // set light switch to position on
            LightSwitch = automatic // or set light switch to position automatic
        }
        Run(42)             // let ECU run for 42ms
        Headlight == on     // check, if the light is on
    }
}
```

Code 3 Test case with alternatives

Every abstract tCF test case is in tCF (*ITestCase*) backend exploded to test cases without alternatives (*ISingleTestCase*) and those test cases are further processed in the generators of the execution platform specific tests.

Multiple blocks of alternatives can be used in series in one test case and then the amount of specific test cases described via one abstract test case can rapidly grow, as all combinations are generated.

GENERATION OF EXECUTION PLATFORM SPECIFIC TEST CASES

The next step is to translate the abstract tCF test cases so they can be executed on various platforms. Therefore, it is necessary to provide a mapping from abstract actors to target specific objects. The software design of various mappings in tCF is shown in *Fig. 4*.

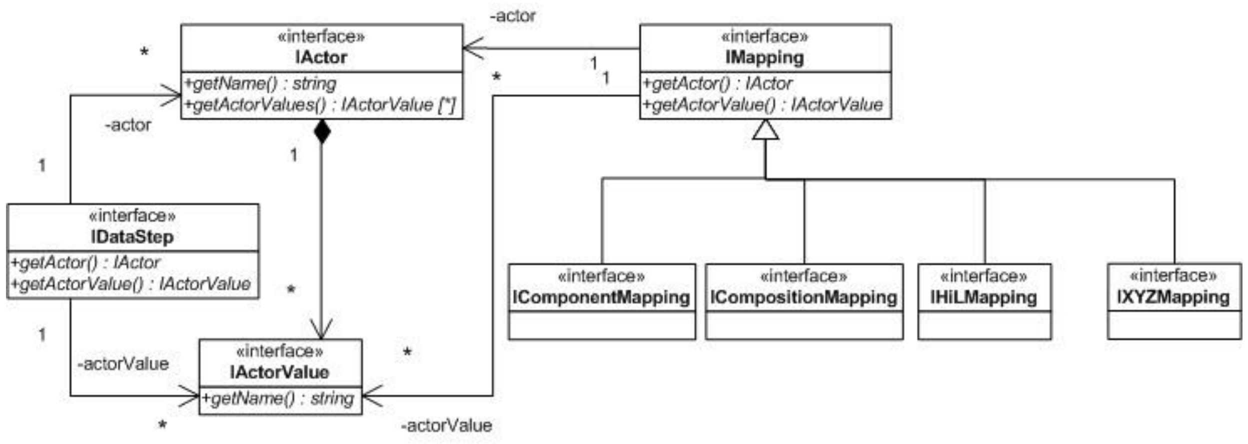
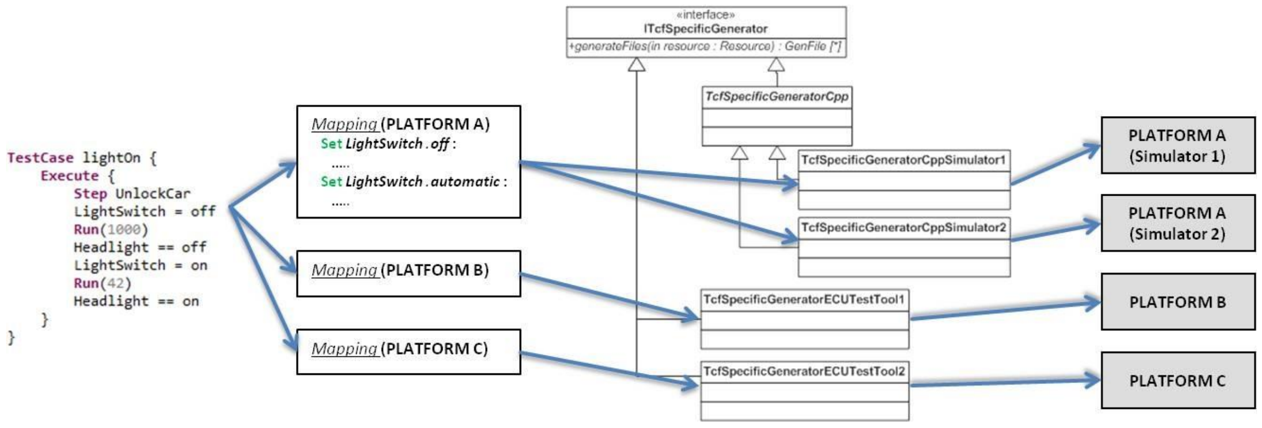


***Fig. 4** Mapping actors to target platform*

Every value of an actor can be mapped to many targets. Therefore many mappings can be defined for every actor. The generator always chooses the right mapping.

The generator is a part of tCF, which generates execution platform specific test cases. There are currently multiple generators for software simulated targets and also for tools performing tests on the target ECU available. *Fig. 5* explains the connection of tCF test cases, their mappings, generators and execution platform specific test cases.



```
TestCase lightOn {
    Execute {
        Step UnlockCar
        LightSwitch = off
        Run(1000)
        Headlight == off
        LightSwitch = on
        Run(42)
        Headlight == on
    }
}
```

Fig. 5 *Generating platform specific test cases*

Mappings must be specified for all actors' values and for all used target platforms. tCF currently contains three software simulated target generators with C++ output, one generator for the target ECU with XML output and one generator for the target ECU with Visual Basic output.

All generators are available in an Eclipse based version and there is also a headless version provided for execution from command line (mainly used in continues integration process).

CONCLUSION

We have introduced basic the part of tCF DSL and its corresponding features. There are many implemented and not mentioned features, as their influence is not crucial for this article. The tCF abstraction helps to develop specification-based tests in high quality and efficiency. The tCF test cases are well maintainable and self-explaining. Execution of the same test case on multiple platforms gives the opportunity to discover hardware errors and incompatibilities. Due to these properties, tCF became a popular and standardized tool in our institution.

Moreover, the explained abstraction makes it very well suited for the combination with model-based testing tools.

REFERENCES

1. BAZIUK, W. BNR/NORTEL. 1995. Path to Improve Product Quality, Reliability and Customer Satisfaction. In: *Sixth International Symposium on Software Reliability Engineering*, pp. 256-262. ISSN 1071-9458
2. ECLIPSE FOUNDATION. Xtext 2.5 Documentation. Available on internet: http://www.eclipse.org/Xtext/documentation/. [accessed: 2014-01-30], 2013.
3. NAGY, R. 2013. Use-Case-basiertes Testen von Softwarekomponenten in der Automobilindustrie. In *OOP - software meets business*. Munich.